Design for Manufacturability: a Feature-Based Agent-Driven Approach.

Dominique Jacquel
Ph.D.
University of Edinburgh
2000

Abstract

The increasingly competitive global market creates the challenge for a faster transition from concept to finished product. This challenge is largely won or lost during the early design stages. Designers require efficient validation tools in order to meet the manufacturability criteria that are expected from them. The adoption of a feature-based design (FBDS) system helps generate mechanical models that are more complete, robust and ultimately more manufacturable. Using the emerging multiagent paradigm at the design-feature level can bring new functionality and increased flexibility to Computer Aided Design and Manufacturing (CAD/CAM) systems.

This thesis presents a feature-based design system called **MADSfm** (**M**ulti**A**gent **D**esign **S**ystem **f**or **m**anufacturability), which allows the creation of 2½D mechanical components, performs continuous manufacturability analysis and solves common problems automatically. The system uses the multiagent paradigm at a feature level to create a new type of active product model. Indeed, each feature inside the product model is embodied by an autonomous agent capable of communicating with its peers, building an image of its world, assessing its fitness in this world and modifying its own geometry to guarantee its manufacturability. Feature agents' activity is based on pre-defined feature validation rules and template solving behaviours. A space partitioner service agent is added to the system that limits the global communication load by ensuring localised propagation of change. Consequently, most activity can take place locally for each individual feature making the approach robust and a good candidate for parallelisation and distribution. The system's design manages to contain problems of stability and communication load. However, problems of activity livelock remain partially unresolved.

The agent-driven approach to feature-based design and manufacturability analysis ensures robust manufacturable designs with a shorter lead-time resulting in substantial cost savings.

Acknowledgement

After three and a half year of hard work, time has come to finish this thing up and thank the people that made it possible. This might just be the most difficult section to write because it should be complete without looking like the yellow pages.

My first thanks go to Karen who supported me, mostly morally (but also financially!) during this project, forced me to take regular vacation in the sun, and married me ☺.

Thank you to Jonathan Salmon for his relaxed supervision that perfectly suited my independent working habits. All our informal discussions helped me more than real meetings could have. Thank you also to the rest of the Manufacturing Planning Group: Frank, Steve, Andy and Greg for their valuable contributions and enjoyable company. Thanks to the entire School of Mechanical Engineering for a friendly and caring environment.

I should not forget all my friends in Edinburgh who shared these few years at the pub, restaurant, cinema and the rest of it. Thank you Satu, Asko, Kook-hee, Al, Tom, ... and the others.

Pour finir, merci à tout ceux qui sont restés en France. Emmanuel, Julien et toute la famille. Merci aussi à Jean-Christophe, Sophie, Jean-Louis, Cyril et les autres pour votre soutien, vos e-mails et avoir fait exploser ma note de telephone! ☺. Désolé, il faudra apprendre l'anglais car il n'y aura pas de version francaise de cette these!

Contents

List of figures

List of tables

List of listings

Glossary

ACL	Agent Communication Language
AFR	Automatic Feature Recognition
AI	Artificial Intelligence
ASV	Alternative Sum of Volumes
B-rep	Boundary representation
BDI	Belief Desire Intention
CAD	Computer-Aided Design
CAM	Computer-Aided Manufacturing
CAPP	Computer-Aided Process Planning
CFD	Computational Fluid Dynamic
CNC	Computerised Numerical Command
CSG	Constructive Solid Geometry
COM	Component Object Model
CORBA	Common Object Request Broker Architecture
DCOM	Distributed Component Object Model
DDE	Dynamic Data Exchange
DFM	Design For Manufacture
EFI	Explicit Feature Interactions
FBDS	Feature-Based Design System
FBM	Feature-Based Model
FE	Finite Element
GUI	Graphical User Interface
IFI	Implicit Feature Interactions
ISO	International Standard Organisation
KQML	Knowledge Query and Manipulation Language
KS	Knowledge Source
MAS	MultiAgent System
MADS	Multiagent Design System
MADSfm	MultiAgent Design System for manufacturability
NC	Numerical Command
OO	Object-Oriented
OOP	Object-Oriented Programming
PLIB	Part LIBrary
STEP	STandard for the Exchange of Product model data

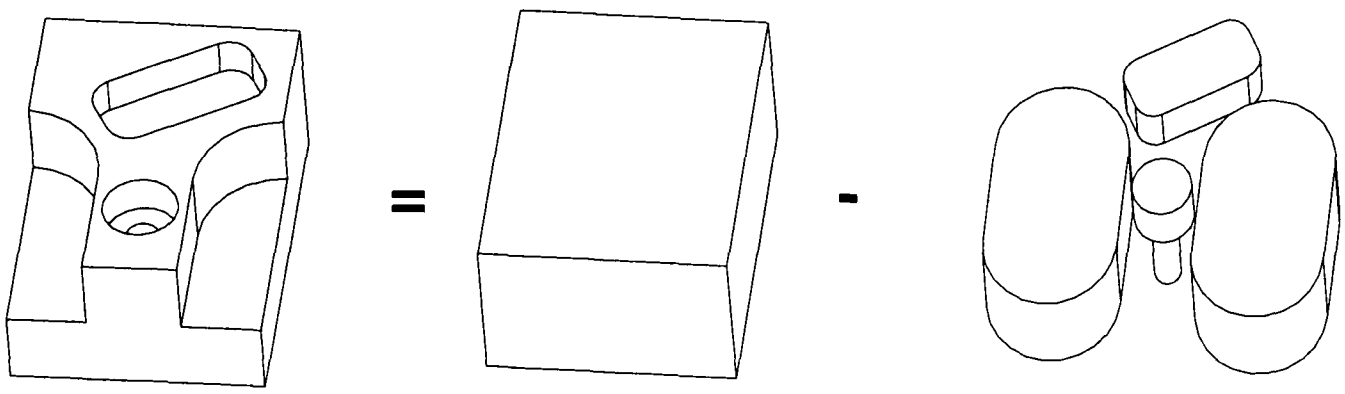
Chapter 1

Introduction

1.1 Background

This chapter discusses the background knowledge essential to the work presented in this thesis. Concepts relating to features in design, manufacturability, process planning and agent technologies are introduced.

1.1.1 Design with Features



(a) finished part (b) blank block (c) machining features removed

Figure 1-1: Design by Feature example

A feature in design can be informally defined as a characteristic of a product that is used to interpret that product in a given domain. For example, in the machining domain, an engineer can interpret a design as a set of material removal operations to be carried out on a blank in order to obtain the desired geometry as shown in Figure 1-1. Another engineer in the design

domain could use a different set of features to interpret the same shape in his particular domain. This thesis, however, focuses on designs based on machining features.

Two different approaches exist for applying features to mechanical design. Feature recognition is discussed in section 2.3 and aims to extract features from existing geometry automatically or semi-automatically. Alternatively "design by feature" lets the designer use a library of form features as building blocks to create the desired component. Two types of features are commonly defined in these libraries:

- form features that usually embody some functional characteristic and are only weakly connected to manufacturing,

- machining features that translate easily into machining operations.

Feature-based design is presented in detail in section 2.4. A potential conflict exists between the principle of design by feature and those of feature recognition. Yet, a hybrid system may be of greater benefit to the user. This thesis' particular interest, however, lies in the Feature-Based design and the possible extensions that it can accommodate. Figure 1-1 shows how a finished component can be expressed in term of positive features (matter) and negative features (material removal) realised through machining.

1.1.2 Process Planning and Manufacturability

A modern mechanical design is typically digital, three-dimensional geometric data representing a part to be manufactured. Elaborate rapid-prototyping machines using stereo-lithography, or other material deposit techniques exist, that can almost transparently use a 3D input files to generate physical objects. However, for production using more traditional machining techniques, this three-dimensional information needs to be translated into a meaningful list of operations, which have to be realised in order to obtain a part with the desired geometry. The list of operations contains machine set-ups, fixture configurations and sequences of cutting operations. It constitutes a process plan. Automated process planners exist **[1, 2, 3, 4, 5]** but one can only produce a workable plan if the desired geometry is not inherently impossible to build given the machining processes used. The need for manufacturability validation tools is clear.

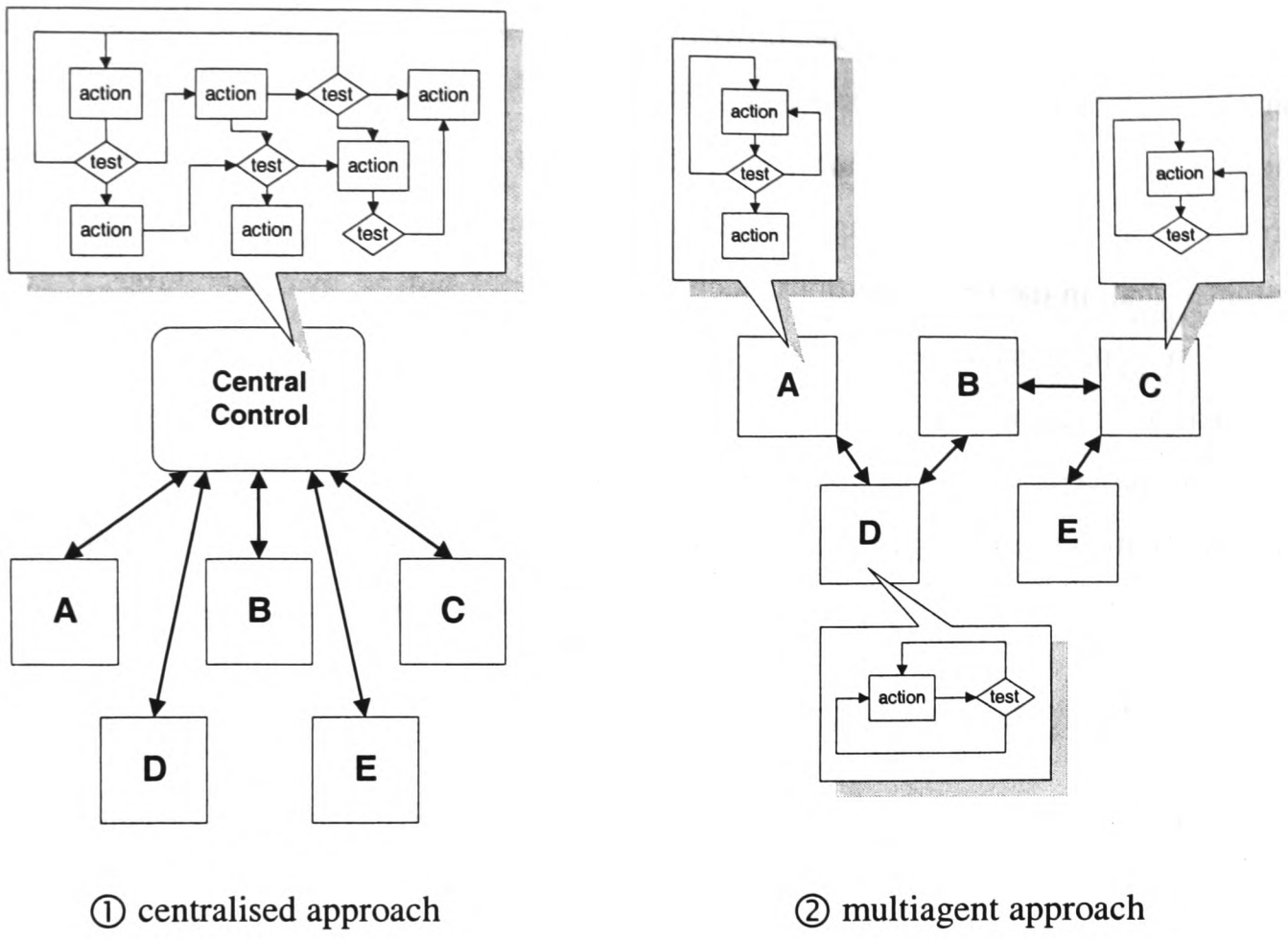
Traditional design techniques can produce product geometry that is impossible, too difficult or too expensive to manufacture. In order to win the "time to market" race, such problematic designs must be detected and dealt with as soon as possible in the product's life

cycle. In this respect, features have proved a valuable and powerful tool to improve product manufacturability. The feature-based approach to mechanical design partially solves manufacturability problems since the set of features used in a system can be chosen to correspond to basic machining operations (micro-cycles). Therefore, it is possible to guarantee that, in the best conditions, each building block will be manufacturable. However, these best conditions cannot be easily guaranteed in a functional design. For example, geometric interactions between features in the model can render the part inappropriate for machining processes. It is necessary to perform geometric tests on the designed model to validate its manufacturability. This reasoning should ideally take place inside the design system so as not to waste resources planning impossible designs.

1.1.3 Multiagent technology

The field of artificial intelligence has recently seen the rise of the new paradigm of *agency* **[6, 7, 8]**. Agents are entities (physical or otherwise) capable of sensing their environment, communicating with their peers and performing actions autonomously in order to achieve their goal(s). Most importantly, agents are capable of collaborative working, as a team, inside a multiagent system. The capacity of agents to co-ordinate their efforts and self-organise into a global system makes them a powerful asset in numerous potential applications.

When dealing with complex problems possessing naturally occurring parallelism or distribution, traditional programming techniques can encounter problems. The complexity of such systems can grow exponentially. Centralised control quickly becomes difficult to handle with the combinatorial explosion of potential interactions. Moreover, using traditional techniques, the behaviour of the system has to be carefully planned which can be difficult when dealing with numerous interacting entities.



① centralised approach ② multiagent approach

Figure 1-2: Centralised and Multiagent Architectures

Multiagent systems are perfectly adapted to tackle situations where interacting entities must react to changing circumstances **[9]**. Figure 1-2 shows two typical control architectures for such a system. Figure 1-2① presents the centralised approach with its inherent bottleneck and the high complexity of its central operation. Figure 1-2② shows the same system controlled using the multiagent architecture. The multiagent approach doesn't rely on a central point for co-ordination. Instead peer to peer interactions are used as a distributed control mechanism. Moreover, the system's global complex behaviour does not need to be completely planned. Interaction between simpler local behaviours in each agent leads to an emergent behaviour in the system **[10, 11]**.

1.1.4 The Feature as Agent approach

Feature Based Design Systems (FBDS) and manufacturability validation tools are already a reality both in research and the commercial world **[12]**. However, existing systems are based upon passive data structures (features) processed by external software modules such as a modeller or a process planner. As seen earlier, the centralised architecture used in these feature-based design systems inherently leads to complexity in the algorithms used and can

create operating bottlenecks. The feature-based approach clearly embodies characteristics that make it a good candidate for agentification. Features are by nature entities involved in geometric relationships with one another inside a product model. It is therefore possible to extend the concept of design features by turning each feature into an autonomous software agent. Such an extension to the concept of design entities brings radical changes to the CAD system's internal structures as well as the functionality offered to the users. The traditional passive product model is transformed into a living community of autonomous agents acting on behalf of the designer. Given feature agents with a sufficient level of skills of each feature-agent the designer can trust the features to carry out model maintenance and analysis tasks, leaving the designer free to focus on real design issues.

1.2 Aims and Objectives

The aim of the work described is this thesis is to investigate the potential of using autonomous software agents to embody design features in a CAD system in terms of system architecture and functionality. Switching from the conventional and/or object-oriented software design paradigm to a multiagent approach represents a major transformation. The purpose is the investigation of the potentials and implications of creating an agent-driven dynamic product model from two points of view. The end user (designer of mechanical parts) perspective needs to be explored to determine how agents can affect the design activity. The architectural and operational repercussions on the CAD system should also be addressed.

Several practical objectives are realised to back up this investigation, which are:

- To experiment with different level of agency and select one able to support a feature-based design system.
- To select a set of feature validation rules of moderate complexity usable for manufacturability analysis. To design solving strategies that can enforce these rules.
- To design and implement a prototype feature-based CAD system using agent technology to provide a 2½D modelling environment for mechanical design.
- To create an active geometric model, powered by autonomous agents, that provides continuous manufacturability analysis and autonomous geometric modification of features.

- To investigate the principle of geometric locality. To design a space-partitioning scheme that permit local operation of feature agents.

An experimental implementation of a feature-based agent-driven system is realised that provides the novel functionality of an active model working on behalf of the designer.

1.3 Outline of the Thesis

The next two chapters of this thesis provide a review of the state of the art of the domain. In Chapter 2, the use of feature in CAD/CAM is surveyed with a particular attention to feature-based design. Chapter 3, discusses manufacturability analysis of product design. The fields of agent technology and multiagent systems are critically presented in Chapter 4. The subsequent chapters form the main body of the thesis and present its main contribution. In Chapter 5, the concept of feature-based agent-driven CAD is explained. Its potentials and shortcomings are discussed. Chapter 6 portrays the experimental implementation work that demonstrates the concept of feature agents. The results gained from this implementation are discussed in Chapter 7. Finally, Chapter 8 summarises the conclusions and surveys an array of potential further research.

Chapter 2

Features in Design and Manufacturing

2.1 Introduction

This chapter consists of three sections. The first section introduces the concept of features in the domains of mechanical design and manufacture. Section two presents an overview of the feature recognition literature (feature recognition per se is outside the scope of this thesis). The third section discusses feature-based design and includes a comprehensive survey of the domain.

2.2 What is a feature?

The Oxford English dictionary has this definition for the word feature: *"a distinctive or noticeable quality of a thing"*. A feature is therefore a quality or characteristic that one can notice in a thing. The Oxford dictionary also defines the verb *interpret* as "*[the act of] understanding in a specified way*". Hence, extracting features from a thing is interpreting it in a given way. The way to understand that thing depends on both the interpreter and the context in which it is interpreted. It seems obvious that one will probably notice different features depending on one's background, profession or particular skills. The interpretation can also be influenced by the context in which it is conducted. For example, an abstract metallic shape in a museum would be interpreted in terms of artistic features such as style and power of expression. However, the same object in an engineering bureau could probably be interpreted, radically differently, in terms of physical properties or functional features.

The different features extracted from the same object sprout from their usefulness in their context. For example, there is little need for a museum's visitor to calculate the centre of gravity of an object on exhibition.

While correct, the definition of feature from the Oxford dictionary does not include the notions of interpretation context and usefulness. We propose to extend the definition to best suit our understanding of what a feature is. A feature becomes "*a distinctive or noticeable quality of a thing useful for interpreting that thing in a given domain*".

2.2.1 Engineering features

Our domain of interest is the design and manufacture of mechanical components, which necessitate the interpretation of physical objects, or their abstract representation (solid models), in term of relevant features. Several definitions have been proposed for features in engineering domains.

"*A feature is any geometric form or entity that is used in reasoning in one or more design or manufacturing activities*" **[13]**.

"*Features are defined as geometric and topological patterns of interest in a part model and which represent high level entities useful in part analysis*" **[14]**.

"*By features we mean the generic shapes or characteristics of a product with which engineers can associate certain attributes and knowledge useful for reasoning about that product*" **[15]**.

Although formulated differently, these definitions indicate a good consensus on what a feature is, for engineering purposes. The common notion here is that features are used for reasoning about the product. That is to say that they are used to capture high-level domain-specific concepts going beyond the simple geometric information. Shah identifies three major lacks of utility and effectiveness in Geometry-based CAD systems **[15]**. They are too low-level for design, which should concentrates on function and behaviour. They do not preserve the designer's intent, making changes time consuming. Finally, geometry-based models do not contain the required information for downstream applications such as process planning. Features compensate for the deficiencies of purely geometric modelling by capturing the high-level concepts engineers need to reason about a product. From our own definition of feature, it is evident that multiple types of features exist for the different engineering domains involved in a product life. This thesis focuses particularly on features

related to the part's geometry that are useful during design, but others exist and are used in different engineering domains. Some important feature types are listed below:

- **Geometric/Form features**: They describe part of a nominal geometry.
- **Tolerance features**: They describe geometric variations from the nominal form.
- **Assembly features**: They describe relationships between parts in an assembly.
- **Functional features**: They describe part of a component in terms of its function.
- **Machining features**: They describe machining operations (usually using cutting processes) to remove material from a component in order to generate desired surfaces.

Our particular interest lies with machining features. For the rest of this thesis, any use of the term "feature" will refer to a machining feature unless otherwise specified.

2.2.2 Machining Features & Process Planning

The usefulness of features in engineering has just been discussed and a number of different feature types have been presented that cover a wide range of activities during the product's life cycle. There is however a critical stage when engineering a new product that occurs at the transition between design and manufacture. Production engineers have to devise a process plan that will allow them to produce the desired geometry using available production techniques. Process planning is a critical phase in a product's life cycle and can benefit from a higher level of automation.

Requicha defines automatic process planning like this: "*Given solid models of the desired part and the raw material, plus tolerancing, material, and other ancillary specification, generate automatically a plan for manufacturing the part and actual instructions to drive NC machines tools and other manufacturing equipment*" **[16]**. The obtained process plan must take into account technological issues such as fixturing problems and machine tools capabilities. In most cases, it must also integrate economical aspects to reduce production costs and increase production capacity. Process planning can be decomposed into the following sub-problems:

- **Feature extraction:**

  A list of machinable features is extracted from the product model that fully defines the finished part.

- **Process selection:**

  A manufacturing process is selected for each extracted feature.

- **Process ordering:**

  Processes are partially ordered to reflect precedence relations between features.

- **Setup planning:**

  The part orientation and location relative to the machine tool is determined. Fixturing is designed to accommodate the part.

- **Tool and machine selection:**

  Tools and machines appropriate for each feature are selected.

- **Process parameter selection:**

  Crucial parameters such as depth of cut, speeds and feeds are selected that will maximise productivity, reduce cost and produce parts of the desired quality.

- **Tool path and NC code generation:**

  Tool trajectories and the actual NC code needed to drive NC machines are generated.

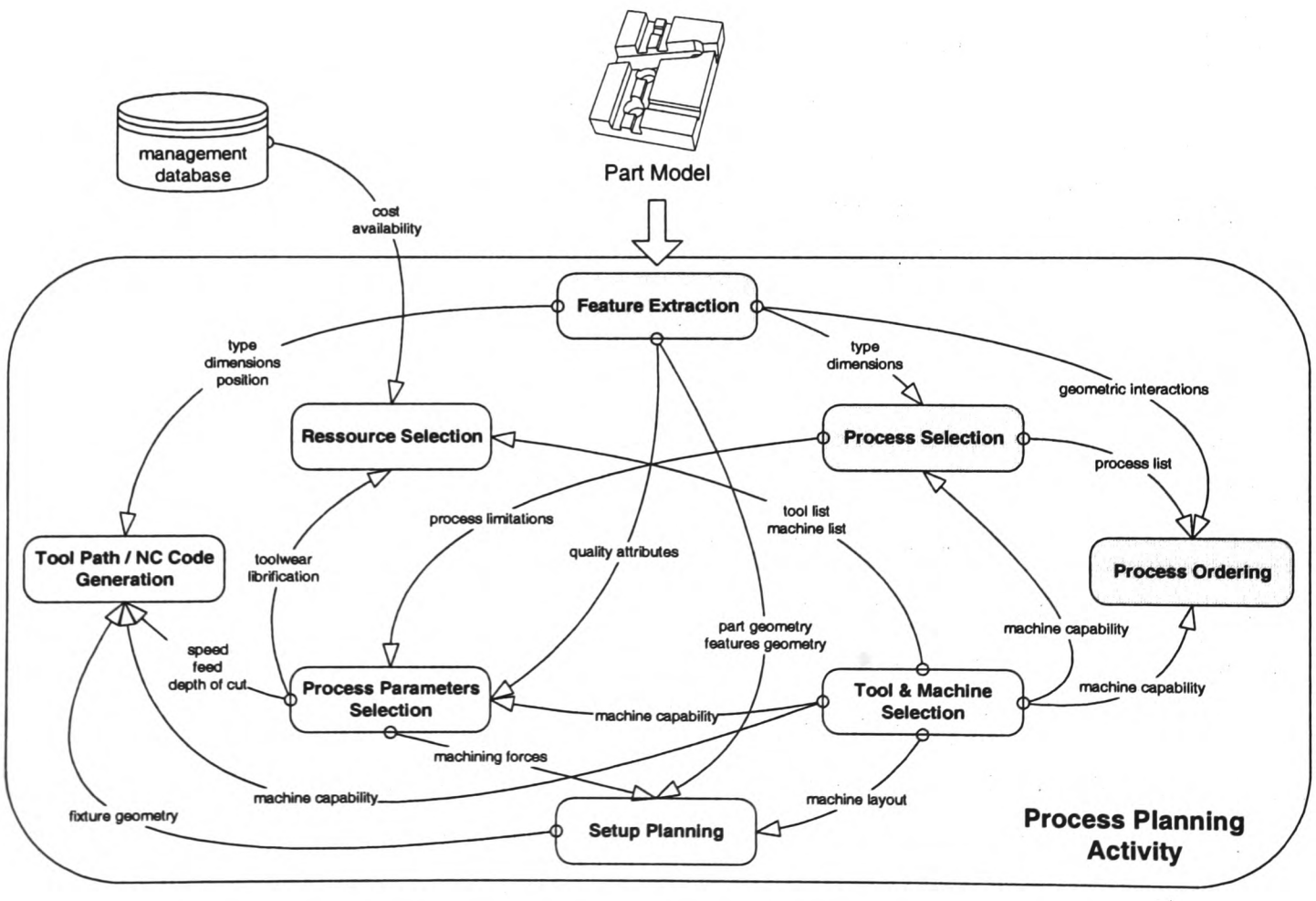


Figure 2-1: Inter-dependence between process planning activities

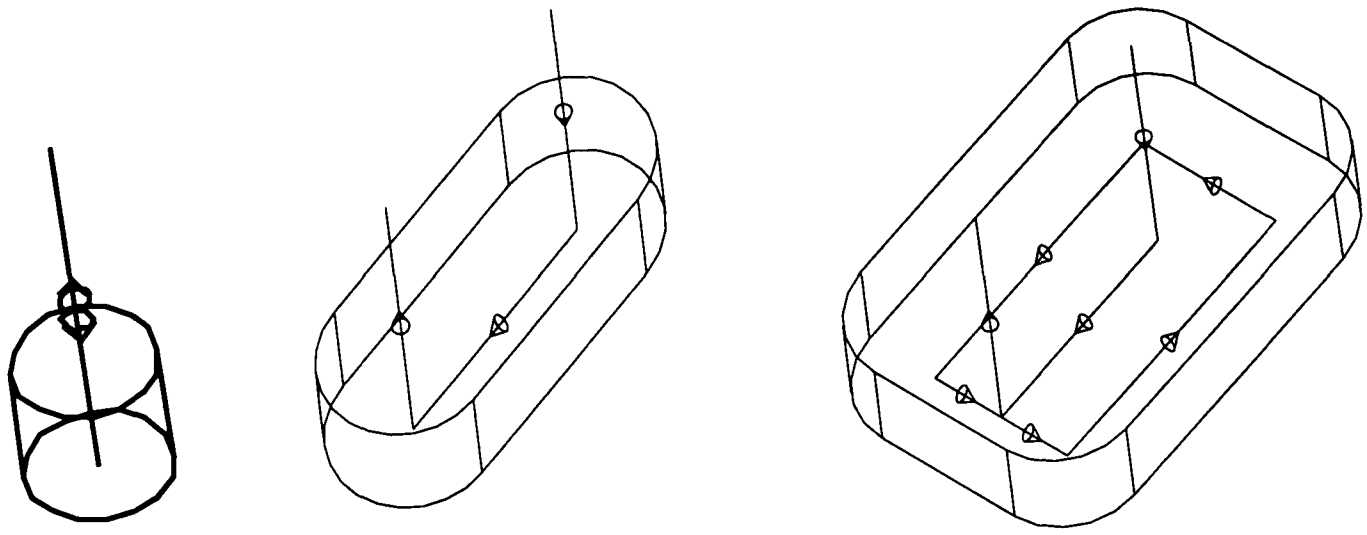
Shah describes process planning as a "*wicked problem*" **[15]**. In particular, he points out the strong inter-dependencies existing between its different sub-problems that makes it so difficult to fully automate. Figure 2-1 illustrates some of the links that exist between the different process planning tasks. Although it doesn't claim to be complete, it helps visualise why fully automated process planning remains a distant dream. It is a complex problem where everything depends (directly or indirectly) on everything else. Two methods are used to achieve some degree of automation in process planning.

- Early automated process planners used the variant approach, which is based on the principle that similar parts will require similar process plans. Variant process planners such as CAM-I **[5]** use part classification to identify group of similar parts. After determining an adequate part group they assist the user in adapting an existing template process plans to suit specific parts.

- More recent process planners generally use the generative approach, which attempts to create process plans from scratch. Generative process planners like PART **[1]**, SIPS **[3,17]** or hutCAPP **[4]** operate on unambiguous geometric models and use geometric reasoning to produce workable process plans.

When dealing with such a complex problem, even localised automation may greatly decrease the difficulty of finding potential solutions. A product model expressed in terms of machining features can facilitate the process planning activity. Machining features provide a context of interpretation for a model that is useful to manufacturing engineers. Sub-activities (pictured grey in Figure 2-1) benefit from a feature model based on machining features. *Feature extraction* is completely eliminated. Moreover, each machining feature can be easily mapped to a manufacturing process therefore greatly simplifying *process selection*. Finally, *process ordering* may also benefit from explicit and implicit precedence information captured by machining features inside the part model.

Our interest lies in 2½D parts that can be produced using cutting processes such as milling and drilling on conventional 3 axis milling machines. Features are realised by material removal using cutting tools rotating along the Z-axis referred to as the ½ dimension. Machining features used in this context correspond to standard machining micro-cycles. Holes, slots and pockets are the most frequent machining features and account for the vast majority of 2½D machining.



① hole ② slot ③ pocket

Figure 2-2: tool path examples for machining features

We can describe these three machining features using two different aspects; the type of surfaces they generate on the component being machined, and also the complexity of the tool path needed to realise them. Figure 2-2 shows the increasing tool path complexity needed to generate these features.

- **Hole:** A material removal operation generating a vertical cylindrical surface. Unless it is a through-hole it also generates a bottom surface that can be flat, spherical or conical depending of the cutter's shape. Using 2½D machining, a hole is realised by feeding the tool inside the material along an uni-dimensional path in the Z direction.

- **Slot:** A material removal operation generating two parallel vertical walls and a flat bottom surface. In the case of a blind or partially blind slot, it also generates one or two vertical cylindrical surfaces at each end. A slot is realised by using a bi-dimensional tool path.

- **Pocket:** A material removal operation generating a flat bottom-surface, several vertical walls and cylindrical "corner" surfaces. The complexity of a pocket depends on the complexity of its XY profile. This defining profile can be convex or concave, and can even use complex curves (spline or NURBS). We restrict ourselves to rectangular pockets generating two pairs of parallel vertical walls and cylindrical surfaces at the corners. A pocket is realised using a 3D tool path.

These three features map with no major difficulties to machining cycles. Template micro-cycles exist that can be used to generate NC code that will control cutting of each feature type. Several templates can exist for a single feature type. Naish shows that selecting the appropriate micro-cycle can be done based on process capabilities **[18, 19]**. In particular, dimension ratios, tolerances and surface finish can be used to select appropriate micro-cycles. The process planner feeds the feature's attributes to selected template micro-cycles and produces a workable plan. Geometric attributes such as dimensions are used to choose appropriate cutting tools and generate tool trajectories. Quality attributes are also used to obtain a final plan. For example, a surface-finish helps determine the cutter speed and feed needed to achieve the desired quality while an inter-feature geometric constraints is used to obtain an ordered sequence of operations.

Ultimately, machining features simplify the process-planning task by allowing a degree of automation in creating the NC code needed to produce a part. Simple mapping between machining features and machining micro-cycles allows the process planner to quickly obtain workable NC programs. The time saved can be used to tackle other problems such as fixturing and material flows.

2.3 Feature recognition

Automatic feature recognition (AFR), *per se*, is not within the scope of this thesis and more complete surveys of the fields are available **[20, 21, 22]**.

One way to adopt features for manufacturability analysis and process planning consists of extracting features from existing geometric models. This approach is very appealing to engineers because it allows all existing designs to be interpreted in terms of features automatically. Models from both geometry-based systems and feature-based systems can be subjected to automatic feature recognition. However, automatically extracting features from a geometric model is far from trivial. In particular, feature recognition systems can encounter difficulties when dealing with complex parts containing multiple geometric interactions between features.

2.3.1 Concepts and Techniques

Kyprianou first investigated feature recognition from 3D solid models **[23]**. Typically, feature recognition systems use either B-rep or CSG model as input parts. Since a Boolean expression may have any number of equivalent expressions, the CSG model of a given part is

not unique. Therefore, B-rep models are usually preferred as they uniquely define the faces of a solid. Since the pioneering work of Kyprianou based on syntactic pattern recognition, different approaches have attracted attention.

2.3.1.a Syntactic pattern recognition

Syntactic pattern recognition was already successfully used in 2D computer vision when it was first investigated for feature extraction on solid models. It is a technique for representing complex patterns in terms of simple sub-patterns and relations between sub-patterns. By recursively decomposing complex patterns into simpler sub-patterns, a vocabulary of primitives is obtained. Grammatical rules are also obtained that describe composition rules, using primitives, for building complex patterns representing features. The approach characterise the overall part shape as the composition of certain geometric primitives. The input syntactic expression representing the part is parsed using grammar rules to identify patterns representing features.

Although successfully applied to 2D problems and axis-symmetric 3D parts (lathed components), the syntactic pattern approach has had limited success with non axis-symmetric 3D parts. The lack of non-ambiguous 3D primitives has been identified as one of the limiting factors **[24]**. Ambiguity in the primitives can lead to one syntactic expression corresponding to several different features. This can lead to erroneous features being identified. Validation rules to filter these wrongly recognised shapes are difficult to derive and not always efficient.

2.3.1.b Graph-based methods

This approach has been the most popular in the feature recognition research community **[25, 26, 27]**. The B-rep model used for input is translated into a graph representing its topology. Typically a graph is made of nodes and links corresponding to faces and edges respectively. Additional information about the model can be added to the graph to represent face orientation or concavity, therefore eliminating some of the ambiguities existing with geometric primitives used in syntactic pattern recognition. Feature's templates are also stored in the system as a collection of graphs. The graph representation of the part is searched, using sub-graph isomorphism, to identify sub-graphs that match the templates of the primitive features. These sub-graphs are later extracted as the features embedded inside the part.

Graph-based methods have proved a powerful improvement on syntactic pattern matching and have been successfully applied on polyhedral objects. It strong point is the ability to

define features using a formal graph grammar. However, both graph construction and search can be computationally expensive. Some methods exist to reduce the search space but these do not increase the domain of recognisable shapes. Moreover, it is accepted that the graph-based approach is typically weak at recognising intersecting features. The template's graph representing features are built from the part topology. Features interactions inside a model make the topological configuration for a given feature non-unique. Although some method exist to recover missing or fragmented faces, it becomes difficult to match a template's graph with the existing sub-graph in the model.

2.3.1.c Rule-based methods

The rule-based methods are based on expert systems and use rules to capture knowledge about features. A rule-based system contains a collection of rules specifying a set of necessary and sufficient preconditions for the patterns found in features. Different rules describe different supported features. A rule is typically a set of conditions describing geometric and topological properties, and a consequence, which is usually the recognition of a feature **[28]**.

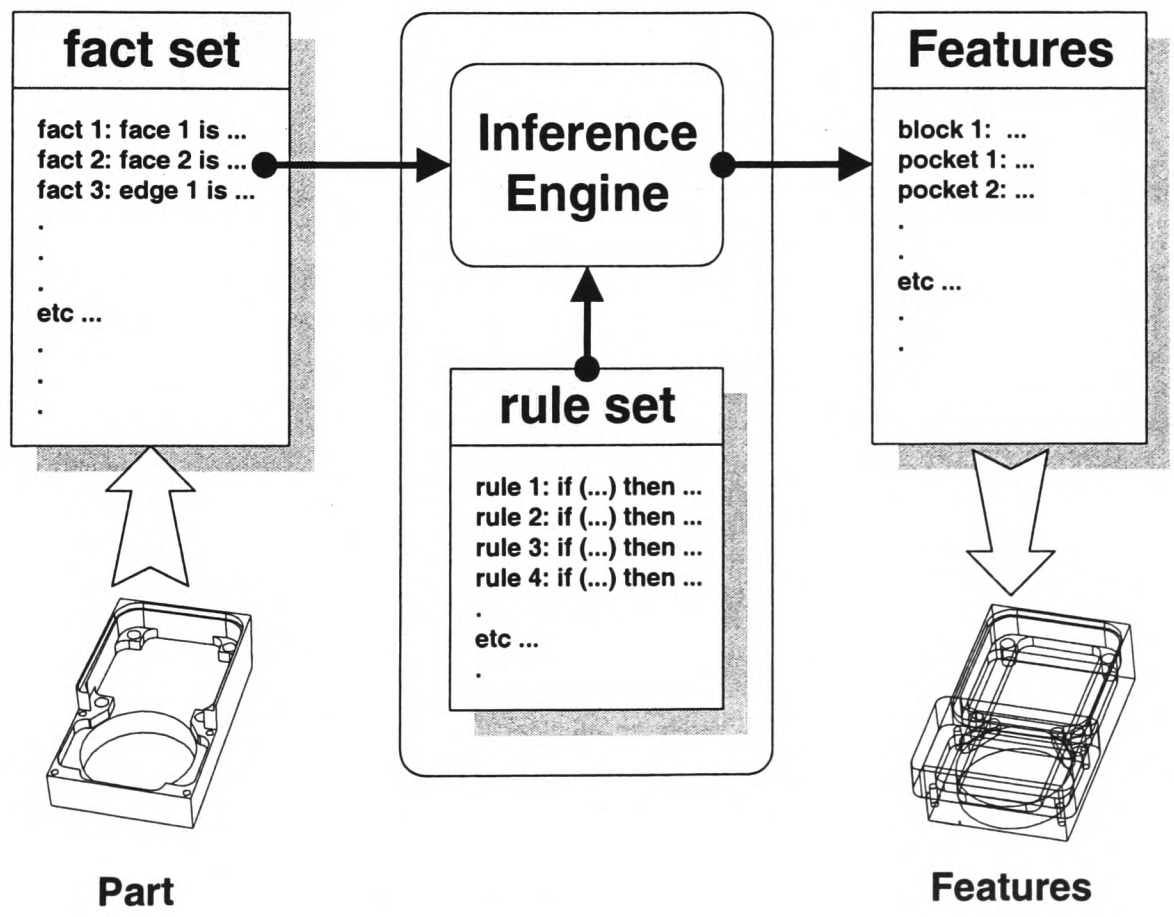


Figure 2-3: Rule-based system operation

Figure 2-3 illustrates the operation of a rule-based AFR system. The input model has to be translated into a suitable collection of facts that are fed to the inference engine. These facts

express geometric and topological information about the part. The engine uses these facts to assess the validity of the rules contained in the system rule set. When the pre-conditions of a rule are verified by the facts provided by the part, a feature is recognised and the engine adds it to its output.

Rule-based systems encounter the same problems as graph-based systems concerning intersecting features. Because intersecting features create non-unique topology, new rules have to be created for every foreseen configuration. Moreover, a rule to recognise a simple feature like a slot might contain large numbers of statements, making the system very verbose.

2.3.1.d Volumetric decomposition

Volumetric decomposition techniques do not rely on properties of surfaces and edges. Instead, these techniques are based on the principle of constructive solid geometry (CSG). They try to find material that must be removed from a blank block to produce a part, also known as the delta-volume. Starting from the desired object, this search is performed recursively until primitive removals are reached. These primitive material removals either represent complete features or can be combined into one. Feature recognition by volumetric decomposition is usually a two-stage process:

- Breaking down of delta-volumes into primitive shapes.
- Feature classification either directly or through recombination of primitive shapes.

Two main decomposition approaches can be defined according to the primitives they use.

Volumetric decomposition : ① **Convex Hull decomposition**

The convex hull of a polyhedron is the smallest convex set containing the polyhedron. In this approach originally proposed by Woo **[29]**, the convex hull is used in conjunction with CSG to decompose a component, in stage, as the difference from its convex hull. This decomposition is also called alternative sum of volumes (ASV). It represents an object by a series of convex volumes with alternate signs **[30, 31]**. Kim solved the lack of convergence for general non-convex parts of the original approach **[32, 33]**. However, problems persist in the case of intersecting features because the shared volume cannot be allocated to all features by the decomposition process. Feature-growing or hint-based recognition must be used to compensate for the lost shared volumes.

Volumetric decomposition : ② **Cell decomposition**

Similar to the convex hull decomposition, the cell-based decomposition uses different geometric primitives when breaking down the delta-volume. The delta-volume is split into minimal cells by extending and intersecting all its surfaces and half-spaces **[34]**. It can be noted that the decomposition only yield a result in the case of a non-convex volumes (convex volumes remain unchanged). The obtained primitive cells usually need to be combined to obtained features. Typically the minimal cells obtained in the first stage are recombined into maximal cells that obey some proximity and adjacency rules. These maximal cells are then combined to obtain final features. The main advantage of this approach is the possibility to generate alternate feature representation of the delta volume by changing the way maximal cells are recombined. However, cell decomposition also introduces new problems. In particular, Han points out that the decomposition extends local geometry to the entire delta-volume and often results in a huge number of cells and calls this problem "*the global effect of local geometry*" **[20]**.

2.3.1.e Hint-based reasoning

We've seen that most methods used for AFR do not handle intersecting features very well. Because intersections alter the face pattern of features, searching for an exact patterns in a part is very likely to fail when intersection occurs **[35]**. To solve this problem and allow for detection of intersecting features, hint-based reasoning (also known as trace or evidence-based reasoning) proposes to process characteristic traces left by intersecting features on an object **[36]**. Hint-based reasoning is typically a two-phase process. First, hints are extracted from the input model. A hint about a feature might be generated from a characteristic combination of faces on the part or by manufacturing attributes contained in the input model. Hints extracted from the part are associated with incomplete features. The second phase is a validation process that will decide if each incomplete feature should be completed and extracted from the part or rejected. The validation process is based on additional geometric, topological and manufacturability constraints. It typically uses a rule-based approach.

Hint-based reasoning is particularly well suited to feature recognition because it is built to handle incomplete, non-unique feature representations. It is also capable of generating alternate interpretation of a part in terms of different sets of features, because identical hints can point to the existence of different features.

2.3.2 Issues in automatic feature recognition

A critical issue in AFR is the ability to recognise intersecting features [20]. Indeed intersecting features are the source of most difficulties in AFR systems. The first consequence of intersections is to render the identification of contributing features more difficult. Intersection tends to alter geometrical and topological configurations inside the model, making it difficult to extract patterns useful for the recognition process. Indeed, a geometric intersection might delete or split faces and edges belonging to the individual contribution of a feature.

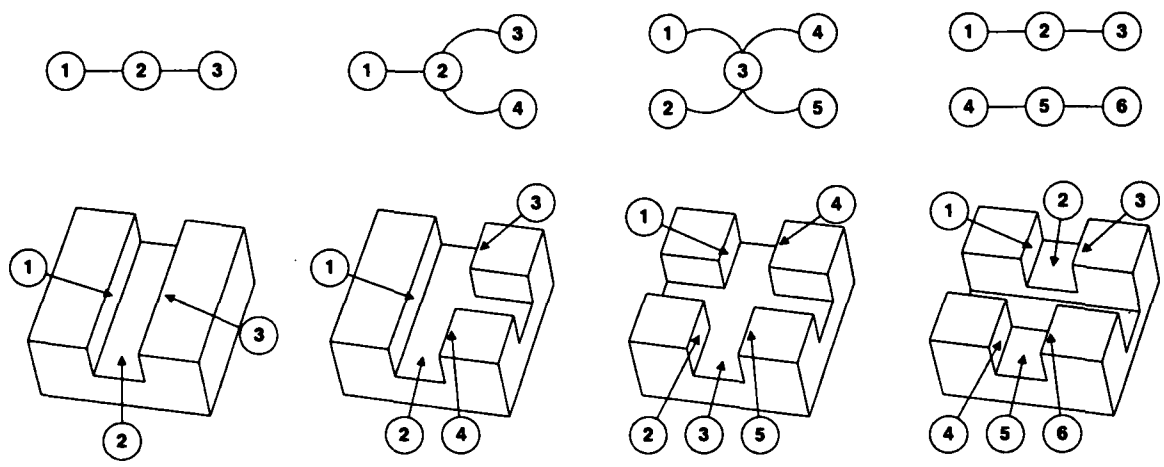


Figure 2-4: Graph representation of an interacting slot

Consider the example illustrated by Figure 2-4. Three different configurations are presented involving intersecting slots on a block. The graph representations (nodes are face, connectors are edges) correspond to the vertical slot in each configuration. It is apparent that the same slot leaves very different traces on the finished components depending on existing interactions, making its recognition a difficult task.

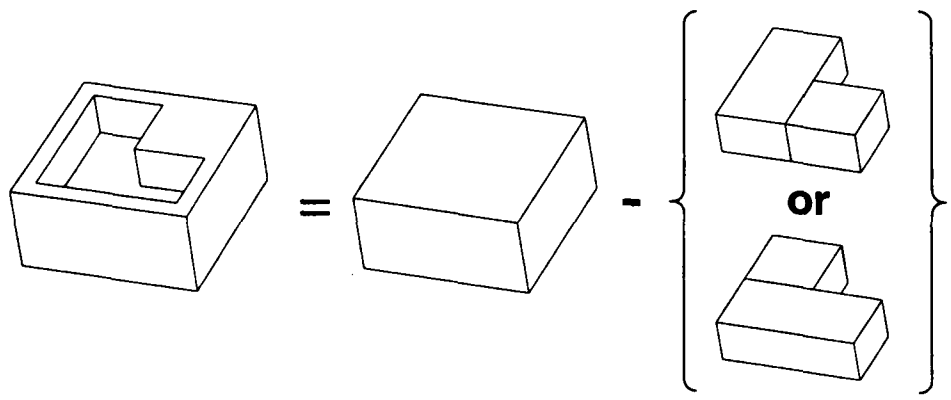


Figure 2-5: Multiple interpretation of interacting features

The other consequence of feature interaction is the possibility of multiple interpretation of a model. Indeed, even the most simple feature interaction may generate multiple interpretation, as shown in Figure 2-5. In this case a simple L shaped pocket is being

interpreted in terms of convex machining volumes and yields two possible interpretations. Although, the possibility to generate alternative feature sets for a part may be of benefit, it also involves the additional decision on which interpretation is best.

2.4 Design by Features

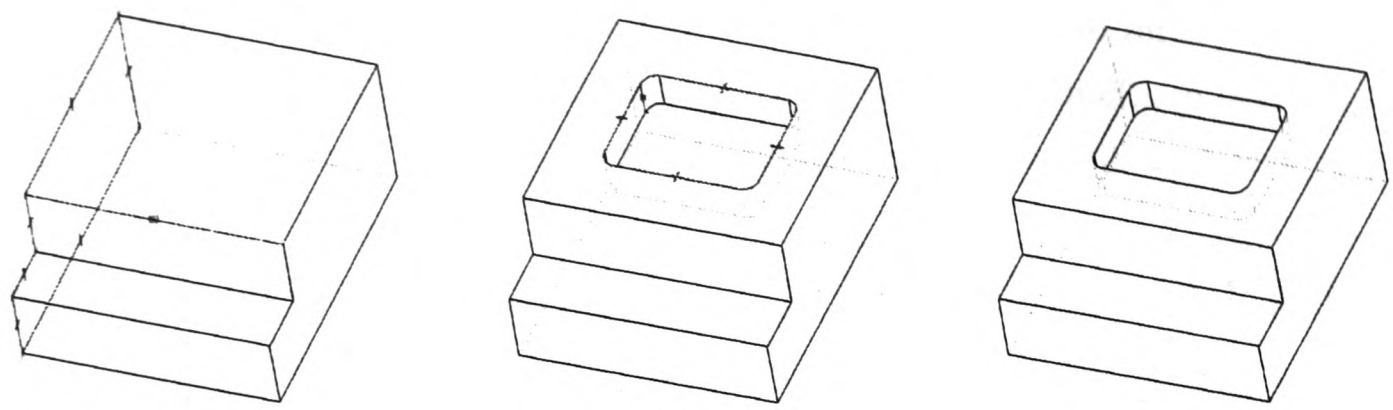
The second popular approach to features in engineering is the design by feature or feature-based design approach. It proposes to introduce features immediately during the design stage. Instead of searching CAD geometric models for features, feature-based design incorporate feature directly into the CAD description.

A critical survey of the literature concerned with design by features is now presented. First, the usefulness of features is considered. Feature taxonomies are discussed and illustrated with classification examples. A comprehensive review of feature creation and manipulation is next. This section covers a wide range of matters: feature library selection, feature attributes, geometric transformations and user-defined feature are among those. The issues of feature interactions inside a model and feature validity are surveyed. Followed by a summary of work on validity maintenance inside feature-based models. Finally, research concerning feature mapping and data exchanges is presented.

The combination of all summarised subsections present a comprehensive view of the field of design by features in mechanical engineering. Particular attention is given to issues concerning feature interaction as well as validity maintenance because they represent the most relevant work for this thesis.

2.4.1 On the usefulness of features

The conventional approach to mechanical design is to generate a nominal geometry of the desired product inside a CAD package using purely geometric techniques. B-rep representations are the most common as they uniquely define a solid and are efficiently computerised. CSG representations also exist despite the non-uniqueness of the produced models.



① extruded profile ② extruded cutout ③ finished part

Figure 2-6: geometry-based modelling

Figure 2-6 shows a possible sequence of operation using geometry-based modelling to create a simple 2½D part. It illustrates how a nominal geometry can be created using conventional modelling techniques. The steps used are as follows:

- A 2D profile of the object cross section is drawn in a 2D view. ①
- This 2D profile is extruded in the third dimension to generate a solid. ①
- A second 2D profile is drawn on the top 2D surface of the solid. ②
- The second profile is extruded inside the solid to create a cutout. ②

The result is a 3D solid representing the nominal geometry of the desired object pictured on ③. Depending on the modelling package used to carry out the described steps, the resulting model might include some feature information or only pure geometric data.

For example, this modelling session might be realised using the ACIS® 3D modelling kernel accessed through the ACIS® 3D Toolkit **[37]**. Listing 2-1 shows the scheme listing used in ACIS® 3D toolkit during a similar session. Creation of 2D profiles is done using an ACIS® ***wire-body*** composed of linear and circular ***edges***. A wire is swept along a defined ***gvector*** to generate a ***solid*** body. Finally the cutout can be obtained by subtracting (with a CSG operator) the second extruded body to the first one. This leads to the generation of a final solid body representing the nominal geometry. However, the model obtained contains no additional information concerning the designer's intent (how it was modelled). Indeed, without additional code in Listing 2-1, the wire-bodies are lost during the sweep operations and the two swept solids are also lost during the CSG operation. Only the final solid body remains and the design would therefore prove time consuming to modify.

```
(define solid_profile
 (wire-body
  (list
   (edge:linear (position 0 0 0) (position 0 20 0))
   (edge:linear (position 0 20 0) (position 20 20 0))
   (edge:linear (position 20 20 0) (position 20 40 0))
   (edge:linear (position 20 40 0) (position 100 40 0))
   (edge:linear (position 100 40 0) (position 100 0 0))
   (edge:linear (position 100 0 0) (position 0 0 0)))))

(define cutout_profile
 (wire-body
  (list
   (edge:linear (position 45 40 20) (position 75 40 20))
   (edge:circular-center-rim (position 75 40 25)
                             (position 75 40 20)
                             (position 80 40 25))
   (edge:linear (position 80 40 25) (position 80 40 75))
   (edge:circular-center-rim (position 75 40 75)
                             (position 80 40 75)
                             (position 75 40 80))
   (edge:linear (position 75 40 80) (position 45 40 80))
   (edge:circular-center-rim (position 45 40 75)
                             (position 45 40 80)
                             (position 40 40 75))
   (edge:linear (position 40 40 75) (position 40 40 25))
   (edge:circular-center-rim (position 45 40 25)
                             (position 40 40 25)
                             (position 45 40 20)))))

(define solid_block (solid:sweep-wire solid_profile (gvector 0 0 100)))

(define cutout (solid:sweep-wire cutout_profile (gvector 0 -10 0)))

(define finished_part (solid:subtract solid_block cutout))
```

Listing 2-1 A modelling session in ACIS® 3D Toolkit

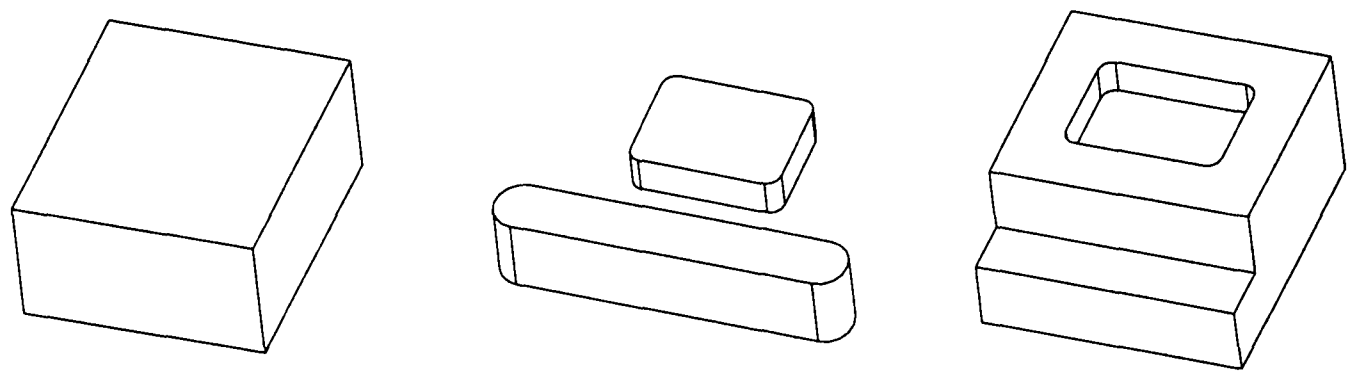
The actual screenshots contained in Figure 2-6 were obtained using SolidEdge™ 5.0 (Unigraphics Solutions™). SolidEdge™ uses the ParaSolid 3D kernel **[38]** and is in fact feature-based. The modelling session followed the same steps as with ACIS® but a major difference exist. The final part obtained is made of two high-level form features called a ***protrusion*** and a ***cutout***. Both are built using 2D profiles and extrusion's vectors. They possess intrinsic materiality used by the geometric modeller to apply CSG operations. They can be modified or deleted at any time. However, they are not hierarchically organised but ordered by creation time instead. This can become a problem with complex part, since a modification of an early feature forces the updating of all later ones. Moreover, the features used here are purely geometric and hold no additional information useful in process planning such as surface finish and tolerances. SolidEdge™ bring the advantages of feature-based design to the designer but does not (yet) offer a strong bridge with process planning and manufacturing. The main advantage of pure geometric features is to preserve some form of

design intent, which can be used for efficient modifications of the obtained geometry. However, it should be noted that, in specific cases investigated by Shapiro parametric solid modellers supporting geometric features can encounter problems that lead to misinterpretation of the real intent of the designer and to erroneous part geometry **[39]**.

In feature-based design, a product model is built by assembling geometric features picked from an existing library. The designer creates a model by creating features and combining them into the model. A feature is chosen from a library of supported feature types, instantiated and specified with dimensions and parameters. Features are then positioned and oriented inside the model using standard geometric transformations (translation and rotation). Two types of design by features can be identified:

- In **destructive modelling** by features, a part model is created by the Boolean subtraction of machining features from a raw stock. This type of feature modelling only allows for the subtraction of machined volume from the existing part and is therefore very efficient for generating process plans. This thesis focuses on destructive modelling techniques.

- The **synthesis by feature** allows both Boolean subtraction and addition of form features from a part. This approach does not require that a raw stock be defined. Although more flexible for the designer, synthesis modelling is less adapted for process plan generation. Indeed, some positive features might have no obvious mapping into manufacturing features. Complex reasoning or feature recognition might be needed for models generated using this approach.



① block creation ② slot and pocket creation ③ finished part

Figure 2-7: feature-based modelling

Figure 2-7 pictures a modelling session inside a system based on machining features. Positive features represent matter (blanks) while negative features represent material to be removed through machining operations. The modelling steps used are described now:

- ① Create a positive block with the required dimensions and attributes.
- ② Create two negative features (slot and pocket) with the required dimensions and attributes. Apply the necessary geometric transform (translation and rotation) to them.
- ③ The finished part is automatically obtained as the result of the solid subtraction of all negative features from all positive features.

Although the nominal product geometry is the same as with geometry-based techniques, the model obtained by design with machining feature now contains information useful for process planning. Indeed it is possible for the designer to add attributes to features. Some attributes such as name label might only be added for convenience but others might be meaningful information needed at later stages of the product's life. In the case of machining features, surface finishes and some geometric tolerances are example of such attributes.

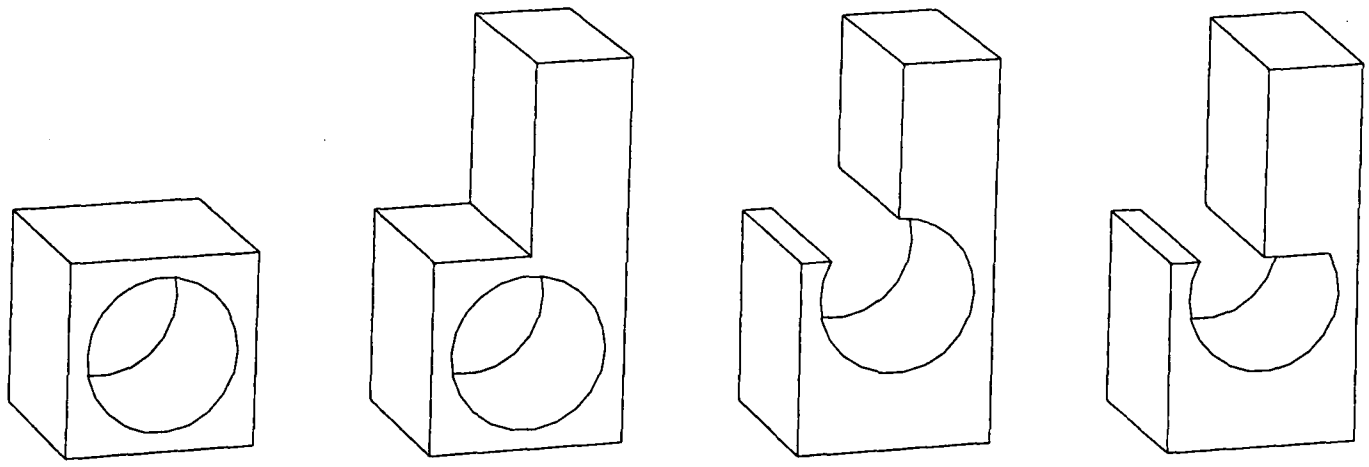


Figure 2-8: interpretation of the designer's intention.

Machining features, like geometric features, permit efficient modification of a model by preserving design intends. Moreover, the use of machining feature in design allows for this capture to be made in terms of unambiguous machining operation that are less prone to misinterpretation. For example, Figure 2-8 borrowed from **[39]** demonstrates how most feature-based parametric modellers fail to correctly retain the designer's intention in the case of a simple hole. The steps followed by the designer are described below:

- ① Creation of a through hole in a block.

- ② A second block in merged with the original block.
- The hole is displaced as to intersect with the newly merged block.

Most feature-based parametric modeller will yield the result shown in ④, which obviously misinterpret what the designer really wants to achieve. Design based on machining feature retains an unambiguous hole-feature regardless of the modifications performed, therefore generating a correct interpretation of the designer's aim (as pictured in ③).

From all the examples given in this section, the usefulness of features during the design stage is clear. Feature-based systems allow capturing of meaningful additional information inside the model. This information preserves the designer's intent, which eases later modifications. In the form of feature attributes, it also conveys important manufacturing concepts to the production engineers.

2.4.2 Feature taxonomies

A feature was defined in section 2.2 as "*a distinctive or noticeable quality of a thing useful for interpreting that thing in a given domain*". The concept of domain of interpretation entails that the number of possible features and feature types is not finite. However, it may be possible to categorise features into classes that provide a degree of independence from the domain of application. Rather than specifying all of the geometrical and topological information that defines a feature for every separate feature type, it is possible to group features into a hierarchical tree structure commonly called feature taxonomy. In **[15],** Shah explains that families of features can be created based on identified properties relating to:

- the type of product they describe (sheet metal, machined, injection moulding)
- the applications they are used in (design, FE analysis, process planning)
- geometry, topology (prismatic, rotational)

The properties of the branch features in the structure are passed down to their leaf features. This hierarchical arrangement reduces the amount of work necessary to add new features to a system. Any taxonomy gives a natural structure for a feature library. It simplifies and encourages the extension of the library. Object oriented techniques are widely used to implement such taxonomies because they make full use of the inheritance relations between feature types.

More than only offering implementation guidelines, taxonomies could play an important role in the integration of different engineering domains. If a generic, domain-independent, feature classification could be agreed between design and manufacture for example. It could become a common language used to describe mechanical parts therefore facilitating data exchanges. Indeed, several classification projects with this objective have been undertaken. Most notable are the ISO efforts in two major standard projects **[40]**. Both STEP (STandard for the Exchange of Product model data, ISO10303) and PLIB (Part LIBrary, ISO13584) contain feature classification efforts.

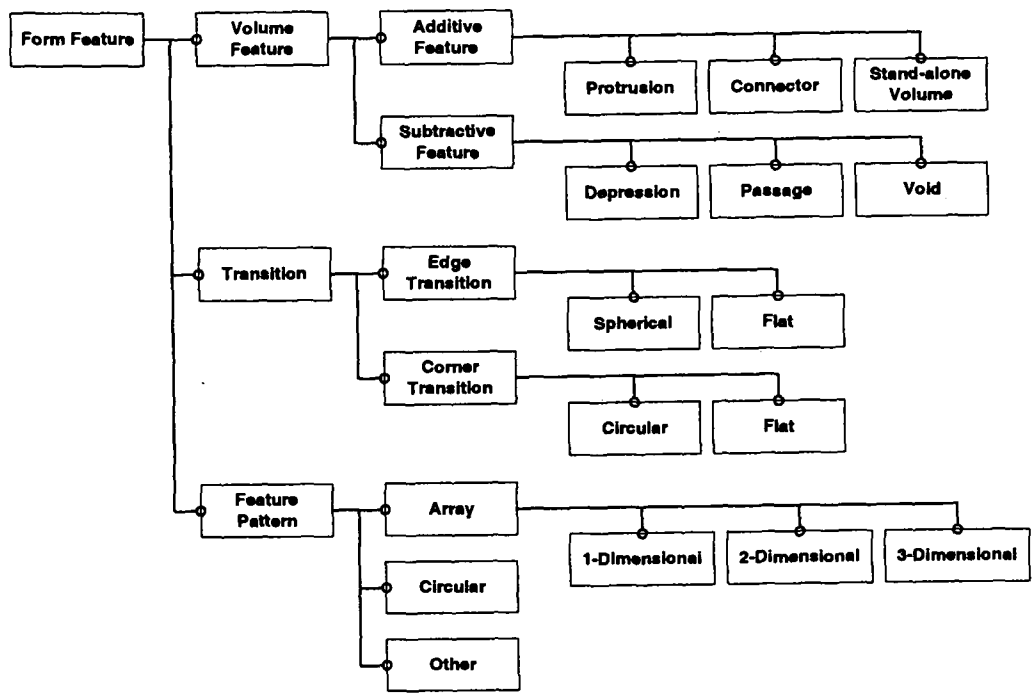


Figure 2-9: Draft STEP form feature taxonomy (ISO-10303-48)

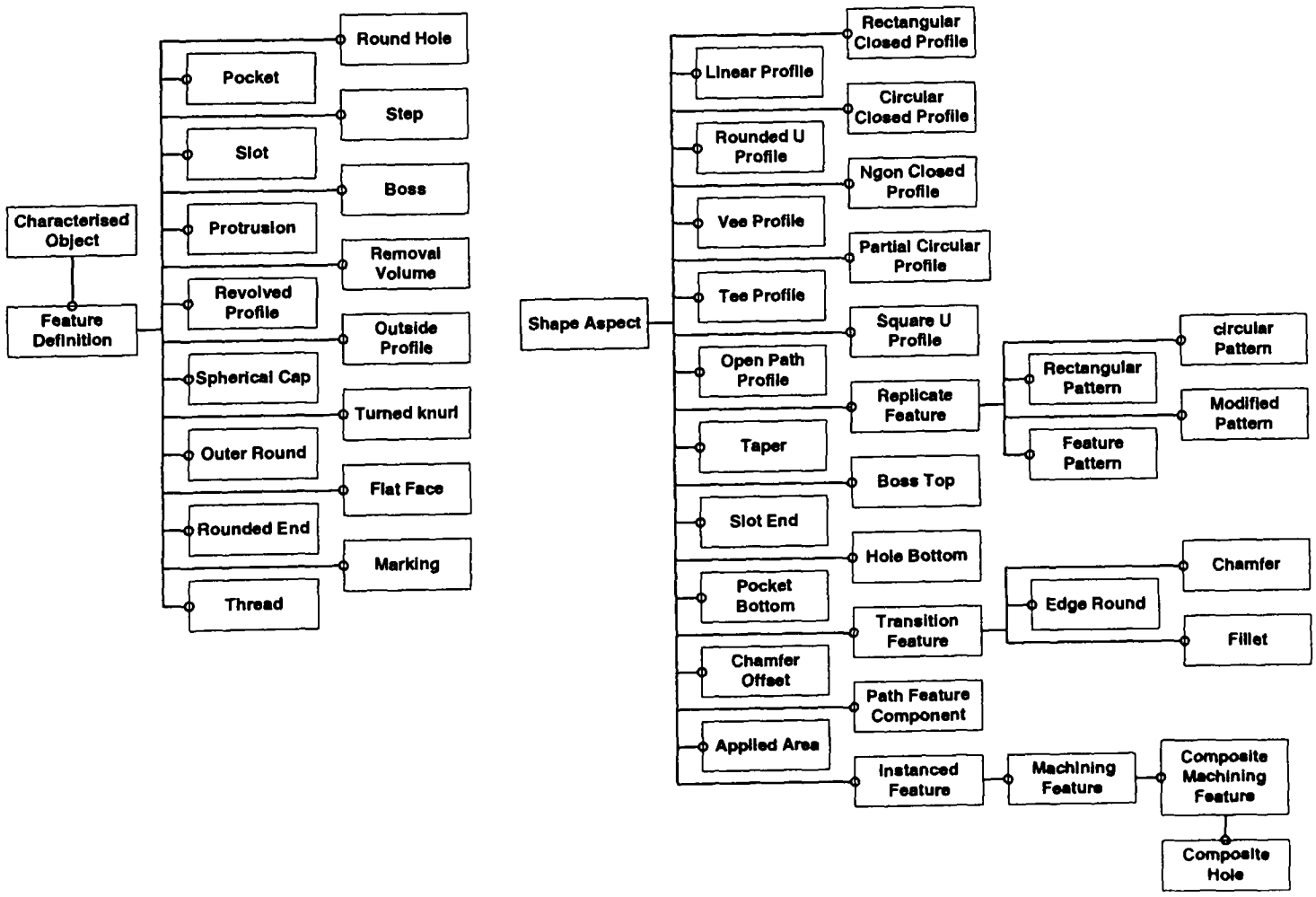


Figure 2-10: Partial STEP process planning features taxonomy (ISO-10303-224)

Inside STEP, two parts were originally devoted to feature description. Part 48 was entitled "Form features" and was to be part of STEP's integrated generic resources. It aimed at providing a library of general-purpose data models representing taxonomy of feature classes as shown in Figure 2-9. The project overlapped and even conflicted with part 224 and was eventually withdrawn before completion. This withdrawal of part 48 from ISO10303 is an excellent example of the difficulties involved in obtaining feature classifications that are generic so to cover a wide domain but specific enough to retain their usefulness.

Part 224 entitled "Mechanical part definitions for process planning using machining features", on the other hand, was completed and will be integrated into the ISO standard. It is pictured in Figure 2-10. Unlike the defunct part 48, it does not intend to be a generic resource but instead provides a specialised resource for process planning using machining features. This taxonomy does not provide a very hierarchic organisation and is highly specialised. In fact, it can be argued that its applicability to other domain is almost non-existent.

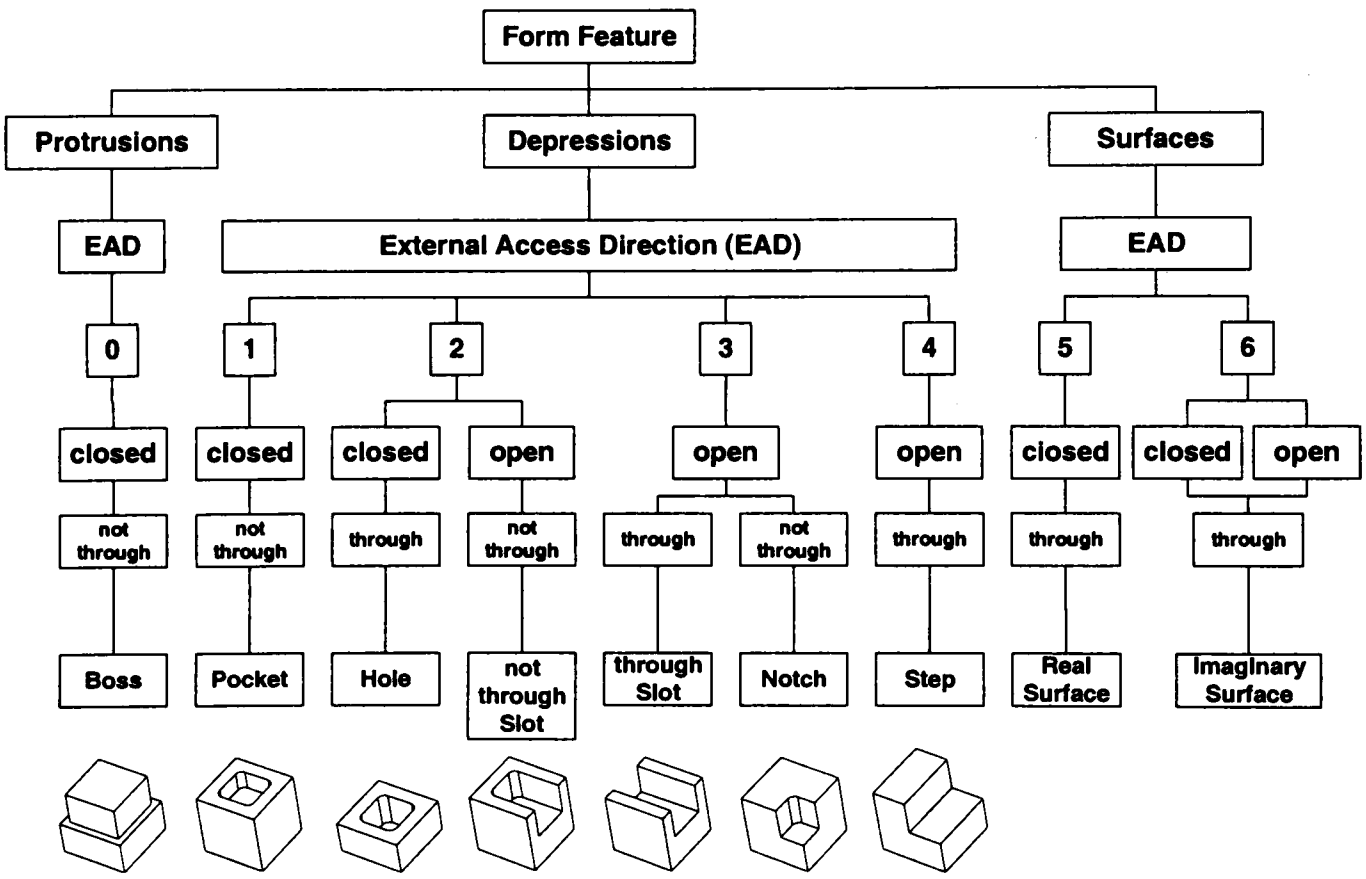


Figure 2-11: Gindy's feature taxonomy

Gindy also proposes an interesting taxonomy (see Figure 2-11) aimed at machining features [41]. The classification process follows 5 stages. Features are first divided into protrusions, depressions and surfaces. In a second stage, Gindy classifies features according to their number of possible external access directions. The feature profile is then used to split

between *closed* and *open* features. A final stage divides obtained feature classes into *through* and *not through* sub-classes. This is an elegant classification, which provides meaningful classes of machining features but is quite specialised.

Other feature taxonomies include those developed for Computer Aided Manufacturing International (CAM-I) in 1985. Pratt and Wilson **[42]** introduce a classification based on explicit and implicit features. An explicit feature is fully defined without calculation while implicit requires its geometry to be calculated. Explicit features are then divided into through holes, protrusions, depressions and areas. Further refinement can be made between prismatic and rotational parts. Butterfield et al **[43]** describes a taxonomy that works by categorising features into three classes: sheet features, prismatic features and rotational features. The sheet features are further categorised as being flat features or formed features. The prismatic features are further categorised as being depressions, protrusions or surfaces. Finally, the rotational features are further categorised as being concentric or non-concentric.

In **[44]**, Bronsvoort points out that creating domain-independent classification of features proves very difficult: "*different applications simply require different features*". This difficulty is illustrated by the withdrawal of the STEP generic form feature classification from the ISO standard. This difficulty should not, however, undermine the usefulness of such schemes in specific domains.

2.4.3 Feature creation and manipulation

One of design-by-feature's aims is to bridge the existing gap between design and manufacture. This demands providing designers with the technological means to generate feature-based representation of parts. Within their design environment, designers must be able to add, modify and delete features inside a product model. This section reviews the concepts involved with providing a feature-based design environment.

2.4.3.a Feature representation

Suitable feature representation must be created to support feature-based design system. A feature representation includes two aspects:

- **Geometric representation** of features is principally based on solid model representation schemes. B-rep features are defined as a grouping of faces, whereas CSG features are defined in terms of two CSG sub-trees of half-space primitives **[45]**. A CSG representation is elegant and easy to edit but is not unique. Moreover, many

applications require a fully evaluated boundary representation, therefore B-rep is the most common representation

- **Knowledge representation** represents the additional information that a feature can hold compare to a purely geometric model. This knowledge is stored as attributes, rules, and procedures.

Feature representation: ① Feature classes and feature instances

A typical mechanical part contains multiple instances of the same type of features. For example, a part may contain many holes. They are all essentially the same but differ by their dimensions, position, orientation and other attributes. It has been seen in 2.4.2 that features families can be efficiently classified in hierarchical structures. Such taxonomies are usually used as a basis for implementing feature-based systems using object-oriented programming **[46]**. A feature library contains feature types implemented as object classes. Designers can choose these classes and instantiate feature objects inside a model.

Feature representation: ② Feature intrinsic properties

Intrinsic properties are also called intra-feature properties. They represent information about the features that do not depend on anything else but the feature itself. They are the core set of values that defined a feature such as:

- **Geometric form**:

  This is the most important aspect of feature definition. In design by feature systems, features are created directly by the user and the desired geometric shape is generated either through a predefined procedure or by solving a set of constraints. Typically, the user chooses a feature type in a library and creates an instance of that class inside the model.

- **Explicit parameters**:

  A feature contains certain parameters that do not depend on other elements of the design and that must be entered during its creation. Usually, the designer explicitly defines some or all of the dimensions a feature. Indeed, some dimensions might be implicitly defined by the feature or be the result of internal/external constraints.

- **Implicit parameters**:

  Some parameters inside a feature might be the results of internal derivation rules.

These parameters are deduced from internal explicit parameters using class-specific rules embedded in the feature.

- **Tolerances**:

  Some tolerances are fully defined within the feature. These might include dimension or form tolerances that do not require external elements from the design to be resolved.

- **Attributes**:

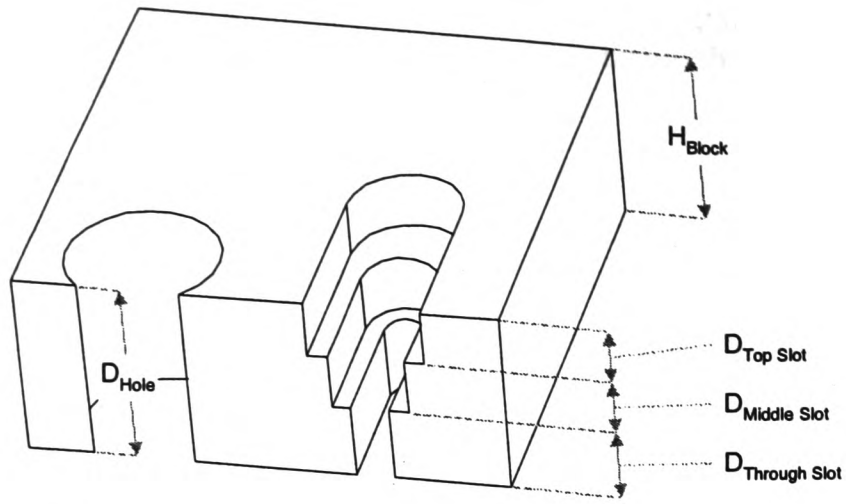
  Other internal attributes might be added to a feature. Simple name labels are commonly added to ease the understanding of models. But other attributes can be added to convey meaningful information about a feature. Subramanian argues that "*attributes are a way for computer-aided design tools, to transfer the non-geometric technical information which is needed for downstream applications in the product life-cycle, onto the CAD model. They are used to represent a wide variety of information, from identifier labels to complex geometric relationships*" **[47]**. For example, a machining feature might contain a required surface finish and multiple others can be added: material name, surface treatment, colour, special process to be used and more. Such attributes are in fact an important aspect of modelling because they can improve the level and quality of reasoning in CAD systems for downstream applications.

Feature representation: ③ **F eature extrinsic properties**

"*Though local, features are not isolated entities. Through geometry and tolerances many kinds of links exist among them*" **[48]**. Extrinsic properties or inter-feature properties are attributes that depend on other elements of the design and cannot therefore be resolved internally. They include:

- **Derived parameters and dimensions**:

  When a feature is created inside an existing model, the designer can define parameters according to existing elements of the design instead of explicitly. Such parameters are referred to as derived parameters. In simple case, the derivation might only involve copying a parameter from another feature. For instance, the depth of a through hole in a block must be equal to the block's height. However, derivation rules can be more advanced and entail complex calculations on parameters from several features inside a model. In the case of nested features illustrated in Figure 2-12, the depth of the through slot at the bottom is derived from the height of the block and the depth of the other slots.



$$D_{Hole} = H_{Block}$$

$$D_{Through\ Slot} = H_{Block} - (D_{Top\ Slot} + D_{Middle\ Slot})$$

Figure 2-12: Derived feature dimensions

- **Position and orientation**:

  In adding a feature to a model, the designer has the option of defining its location by means of entities instead of providing explicit co-ordinates. For example, the nested slots in Figure 2-12 can be positioned and oriented by using existing faces in the model. The through slot is positioned using the bottom face of the middle slot. The middle slot is positioned using the bottom face of the top slot. Finally the top slot has it vertical co-ordinate tied with the block top face.

  The main advantage in using extrinsic positioning is that the constraints between features can be recorded and used to hold the designer's intent.

- **Tolerances**:

  Many geometric tolerances (dimension, orientation) in a design involve more than one features. Dimension tolerances are often expressed between faces belonging to different features.

2.4.3.b Feature creation

Although it might not be fully implemented as such, It is convenient to view a feature-based model as consisting of two conceptually separate components, namely a feature model and a geometric model. The feature model is an abstracted collection of information: feature properties, relationships and other high-level data. The geometric model is contains a low-level geometric representation (B-rep, CSG or other) of the resulting object. The correspondence between these two parallel representations is guaranteed by the system. Several feature creation schemes are possible as shown in Figure 2-13 **[15]**.

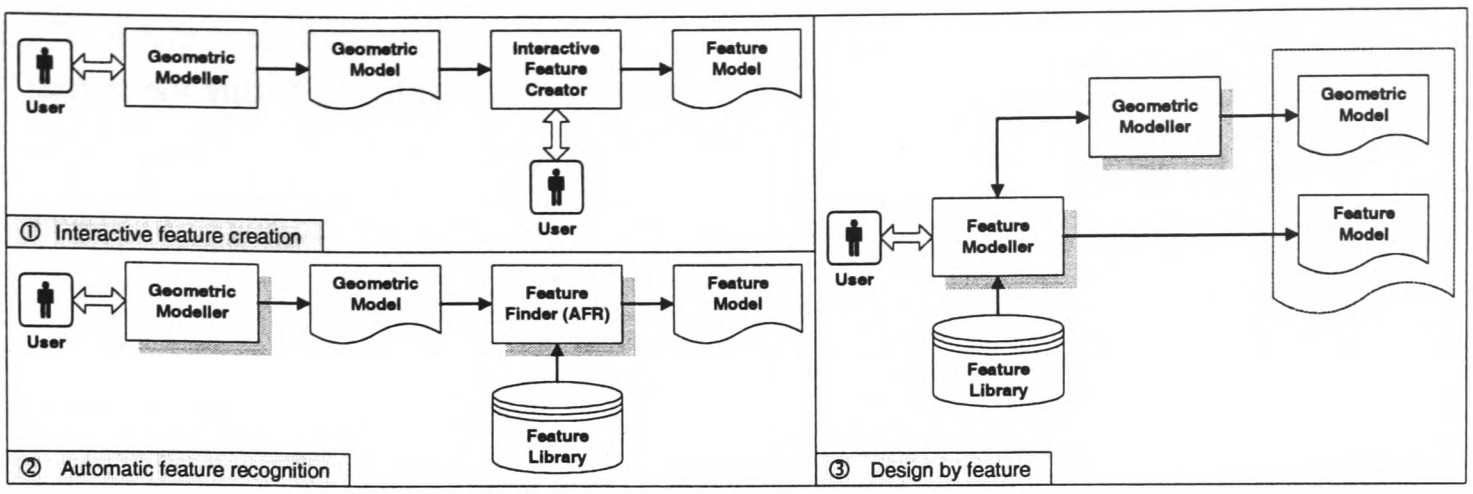


Figure 2-13: Feature creation schemes

Interactive feature creation (Figure 2-13①) is a simple scheme in which the geometric model is created first, with a solid modeller. In a second phase the geometric model is fed to an interactive feature creation application. This application displays the geometric model and allows the user to select geometrical entities (edges, faces) as a basis to construct features. This is of limited appeal because of the weak integration it achieves between CAD and CAM.

Automatic feature recognition (Figure 2-13②) has already been discussed in section 2.3 and replaces the interactive approach (Figure 2-13①) by a fully automated feature extraction application. The only interface with the user is done in the first stage through a solid modeller.

Design by feature (Figure 2-13③) is the creation scheme of interest for this thesis. In this scheme the user interacts with a feature modeller to generate the feature model. A geometric modeller linked to the feature modeller automatically generates the geometric model. The synchronisation between the two models is guaranteed by the system. Shah identifies two possible approaches to feature creation based on the way constraints are handled by the system **[15]**:

- **Procedural creation:**

  In this approach, features are defined by a set of rules and procedures. The procedures provide methods to instancing, destroying and modifying features, creating a solid representation and validating a model. The rules embedded in each feature triggers pre-defined procedures that compute all their derived parameters to reflect their nature. It provides a uni-directional mechanism for change propagation because the order in which procedures are assessed is pre-defined. It hides the constraints away from the user by encapsulating them inside pre-defined procedures.

- **Declarative creation:**

  The declarative approach uses constraint representation to explicitly define the spatial relationship between the different geometric entities forming features. This might mean relationship between primitive volumes or grouping of specific edges and faces for boundary representations. This method offers a bi-directional chain of change in a model. However, it is computationally expensive, as all constraints need to be checked after each change.

2.4.3.c Transformations of features

A fully functional modelling application should allow easy manipulation of existing features inside a model. It requires that defining feature parameters are made accessible to designers for manipulation. Two types of feature transformations can be identified:

- Geometric transforms:

  Modification of features geometric properties lie at the core of the modelling process. Feature modellers provide two levels of geometric transformations. Translations and rotations are used to position features in space and do not affect the shape of features. To modify the geometrical aspect of features, designers can alter feature dimensions directly through their intrinsic parameters. Normal transforms, such a scaling, are usually not used, as they do not maintain features properties. For example, the scaling down of a pocket affect its corner radius, which is not desirable in all cases (see Figure 2-14 ① and ②). Directly changing this pocket dimensions gives better control by allowing maintaining constant corner radius (see Figure 2-14 ① and ③).

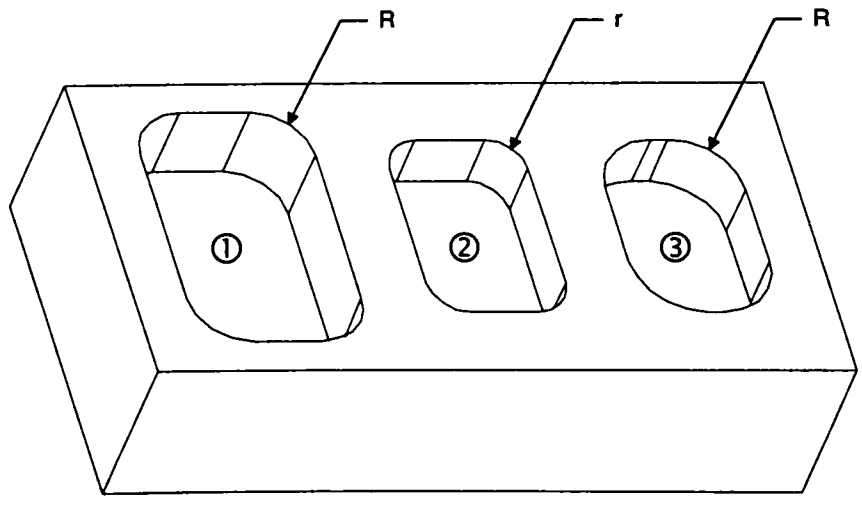


Figure 2-14: Geometric modification, scaling vs. dimension editing

- Attribute change:

  It has been seen in section 2.4.3.a that feature attributes are used to convey non-geometrical information meaningful to downstream application such as process planning. Modification of attribute is therefore an essential means with which designers can reflect changes in their specific requirements. Surface finishes are typical attributes added during design that often require adjusting during process planning.

2.4.3.d User defined features

The need for supporting user-defined features is often expressed by experts in feature-based engineering.

"*Different applications require different sets of features. Moreover, feature sets may depend uniquely on the requirements posed by a particular company and product. Therefore, some type of a facility for extending the feature set of a feature modeller with user-defined features is highly desirable*" **[49]**.

"*Designing mechanical parts using a feature vocabulary is a very effective and rich paradigm. Its expressive power, however, is severely limited if the set of feature types available in a feature library is fixed. It is, therefore, desirable to be able to extend and configure a feature library according to particular requirements*" **[50]**.

User-defined features allow designers to capture their particular knowledge inside custom features, which helps adapting CAD tools to specific needs. Support for user-defined features can be obtained by extending existing feature types or by defining totally new types. The extension approach is simpler to accommodate. In most cases it involves allowing users to add new attributes to supported feature types. It is also possible to use recorded macros to capture sequences of operation that can be replayed to build customised instances of existing feature types. A more difficult approach is to allow designers to define completely new types of features. It is a difficult task, which requires the creation of a new feature type, the definition of necessary parameters, the description of topological relationship between geometric elements, the creation of validation rules and so forth. All these tasks can require advanced programming skills as well as a deep understanding of the specific domain of the created features **[49]**. The absence of support for user-defined features remains a potential weakness of the feature-based design by limiting the scope of modellers to their supported

feature set. However, new declarative approaches are emerging that makes user-defined features a more realistic prospect **[50, 51, 52]**.

2.4.4 Feature interactions

The design of any component aims at capturing functional requirements in terms of a manufacturable geometry. A designer therefore attempts to create the model for a geometric object, which will possess desired physical properties. Using design by feature techniques, the capture of functionality is achieved by creating geometric interaction between features inside the model. Therefore it can be said that feature interactions are the principal means for capturing functionality during the design process. However it is also the main cause of difficulties in terms of model validity and manufacturability. Active research has been pursued in this domain **[53, 54, 55, 56, 57, 58, 59]** and particularly in Edinburgh where feature-based design has been investigated as a requirement for achieving automated process planning **[2, 60, 61, 62]**

In **[60]** and **[61]**, the difficulties presented by the interaction of relatively simple features are outlined by means of examples. The Edinburgh composite component is presented that combines problematic feature-interactions. It is argued that in a machining environment, features cannot be considered individually since the cutting strategy has to take into account relationship between features. Relationships may be due to proximity, overlap, geometric tolerance or other considerations that are important for the production engineer. It is concluded that most difficulties in the production engineering of products are due to interactions rather than the features themselves and that feature interactions must be satisfactorily modelled if true generative process planning is to take place.

Mill classes feature interaction in two types, explicit and implicit interactions **[2]**. Explicit feature interactions (EFIs) are those interactions that are both required and specified by the designer. There are expressed as relational tolerances between features or blanks and they may include concentricity and parallelism. Implicit feature interactions (IFIs) represent knowledge about the component not made explicit by the designer but that is required when considering methods of manufacture. Recognition algorithms for these IFIs are presented that check for a number of problems, namely: void detection, feature presence, access-problem detection, feature intersection, and feature proximity.

Salmon discusses geometric reasoning required to generate anteriority constraints for subsequent process planning of 2½D components **[62]**. The FODDS system described uses

Minkowski sum and medial axis operators to analyse the implicit feature interactions introduced in **[2]**. It generates reports on interaction within a model and detects potential machining problems.

Regli believes that "*how interactions are handled is of vital importance in manufacturing planning*" **[55]**. He attempts to build a general notion of "what are feature interactions?" by presenting different types of intersections and their significance in terms of process planning.

In **[56, 63, 64, 65]**, Bidarra presents validity maintenance mechanism for feature-based models that rely on automatic detection of feature interactions in a model. Bidarra presents different classes of problematic interactions that are uses in his SPIFF system. His interest lies specifically with interaction that invalidates the semantic of features, namely: splitting, disconnection, boundary clearance, volume clearance, closure, absorption, geometric and transmutation.

Faheem makes the distinction between *feature interaction* and *manufacturing interaction* and argues that it is important to distinguish them **[57]**. Feature interactions arise when two or more features geometrically intersect. Machining interactions describe two or more machining operations interfering with each other. For Faheem, although related, both interaction types arise from different phenomena and often need to be addressed at different times using different methods.

In **[58]**, Wong also emphasises the importance of dealing with feature interactions when interpreting geometrical and topological CAD data into manufacturing-specific information. In his words: "*The existence of interacting features is still one of the major problems in both automatic feature recognition and feature-based design. For feature recognition, the presence of interacting features may lead to the possibility of multiple interpretation [...]. In feature-based system [...] multiple interpretation is not an issue. However, interacting features in a feature-based design system can generate invalid features*". Wong presents an approach using machining features for detecting and interpreting feature interactions and an implementation called EXPO. Decomposition into air volumes and material volumes of features is performed, feature validity rules are applied and a final recombination into machining clusters ensures minimum number of machining operations.

2.4.5 Feature validation

It has been seen that features are especially useful because they help capture the designer's intent during the design process. Indeed if the designer decides to create a hole at the centre of a pocket, it is desirable to capture this information instead of merely calculating a position and forgetting the intention that led to that position. This additional information held by a feature-based model becomes immensely valuable when modification must be performed on a design. Imagine that the designer now decides to change the dimensions of the pocket. What should happen to the hole originally placed at its centre? Should it be updated alongside the pocket or should it remain unchanged? Is the validity of the model compromised by the modification?

All these questions need to be answered in order to guarantee the quality of the design. Feature-based design provides an efficient way to hold the information necessary to evaluate the validity of a model during its manipulation. Active research is pursued concerning model interrogation, manipulation and validations. Gindy surveys this area of research in **[66]**. Dohmen also gives a short but good survey of the current approaches to feature validation in **[67]**. The most relevant and most recent work is presented in the following sections.

2.4.5.a Feature manipulation issues

Features are used for a variety of purposes but according to Rossignac, using geometric features may ease expressing and performing engineering changes or simply corrections of design errors. Despite great advances in CAD, design remains an iterative, error-prone process. As early as 1990, Rossignac worked on the idea that features could offer a high-level vocabulary for characterising errors in design and for specifying how they should be corrected. He argues that "*error detection may be automated by supporting intentional features, which correspond to the desired characteristics of the model, and by endowing them with domain dependent validity criteria expressed in terms of associated geometric elements*" **[68]**.

Rossignac views the design process "*as an iterative transfer of information between two models: an intentional model, which captures the functional requirements, and an extensional model, which is a concrete realisation of the functional requirements*". Despite research on automatic synthesis of extensional models (geometry, topology) based on functional specification, the incremental design approach is expected to dominate the years to come. In this context a more intelligent CAD system should provides the user with a

means of expressing its intentions, take care of tedious calculations and provide valuable feedback. Armed with this feedback, the designer can then proceed to modify the design in order to meet functional requirements. A modification made to solve a particular problem can create new ones in the model. The CAD system should be able to efficiently detect these problems in order to accelerate the convergence of the design process toward a solution. In this work, Rossignac always "*assumed that no automated solution exists and that human intervention is necessary to correct the side effect of these editing operations*" **[68]**. His intentional features only serve the purpose of detecting problems.

Intentional features are a mechanism for interrogating the extensional model and evaluate its validity against rules contained in the intentional model. These rules may be intrinsic feature rules or additional constraints added by the designer.

2.4.5.b Self validation features

More recently, Mandorli introduced the idea of self-validating features in **[69]**. He argues that "*a key problem in feature-based modelling is how to maintain the consistent correspondence between the geometrical description of a feature and its related functional meaning (semantics) during the entire modelling process*". This is truly the concept of intentional and extensional models discussed by Rossignac.

Mandorli's intent is to define a feature-based system where each feature is an active entity provided with self-validation capabilities. To achieve this, he creates features as object described by three sets of properties and a set of validation methods:

- Technological properties and Location properties
- Shape properties:

  the shape of a feature is defines using geometric primitives and parameters.
- Validation methods:

  methods, represented in terms of rules, controlling the functional consistency of the feature in respect of the intended feature behaviour.

The validation methods are topological and geometrical constraints applied to some of the shape properties used to define the feature. During the design process, the feature-based system performs geometrical evaluation of the model. The result of these evaluations is stored in attributes attached to the shape properties. These attribute are OLD, NEW, MERGE, SPLIT, CHANGE, REPLACE and DELETE and represent the state of all shape

properties in the model. The validation rules define what values each shape property is allowed to take in order to respect the intended function of the feature. After a modification in the model, the system updates the state attribute of modified shape properties. An unaffected feature will have only OLD attributes attached to its shape properties. Others will have one or more attributes in another state. Each feature can then check the attributes of its shape properties against its validation rules. If a mismatch is detected the system can inform the user that the last change resulted in the feature's validity being compromised.

For example a blind hole is defined as follows:

Shape properties:
primitive_shape = *cylinder*
Bottom = *bottom_face_id*
Lateral = *lateral_face_id*
Entrance = *entrance_loop_id*
Operator = Subtract

Validation methods:
Bottom = *not_delete_rule*
Lateral = *not_split_rule*
Entrance = *convex_rule*

The validation rules are chosen so to enforce the function of the feature, which is to be a blind hole. For example, if the hole were to become a through hole during a modification, the system would update its bottom face to a DELETE state. Therefore, the *not_delete_rule* applied to the bottom face of the hole makes sure the hole does not become through.

Mandorli demonstrate that self-introspection of entities supported by reflective control and associated validation rules is a promising approach allowing a feature to validate its own semantic after each change occurring in a model. However, It should be noted that Mandorli uses the term "active" to signify that validation computations are performed inside features, but his approach does not entail feature autonomy.

2.4.5.c Validity maintenance of interacting features

The latest development in feature validation is the work of Bidarra and Bronsvoort in **[63, 64]** concerning validity maintenance of semantic feature models.

Bidarra notes that "*one of the most powerful characteristic of feature-based modelling is the ability to associate functional and engineering information the shape information in a product model.*" And that "*most systems fail to consistently maintain the meaning of the features throughout the modelling process*" **[64]**. He also deplores that in some systems, features are only present as a user interface abstraction and are not actually present in the

purely geometric models generated. This is an issue that this thesis has previously discussed in 2.4.1.

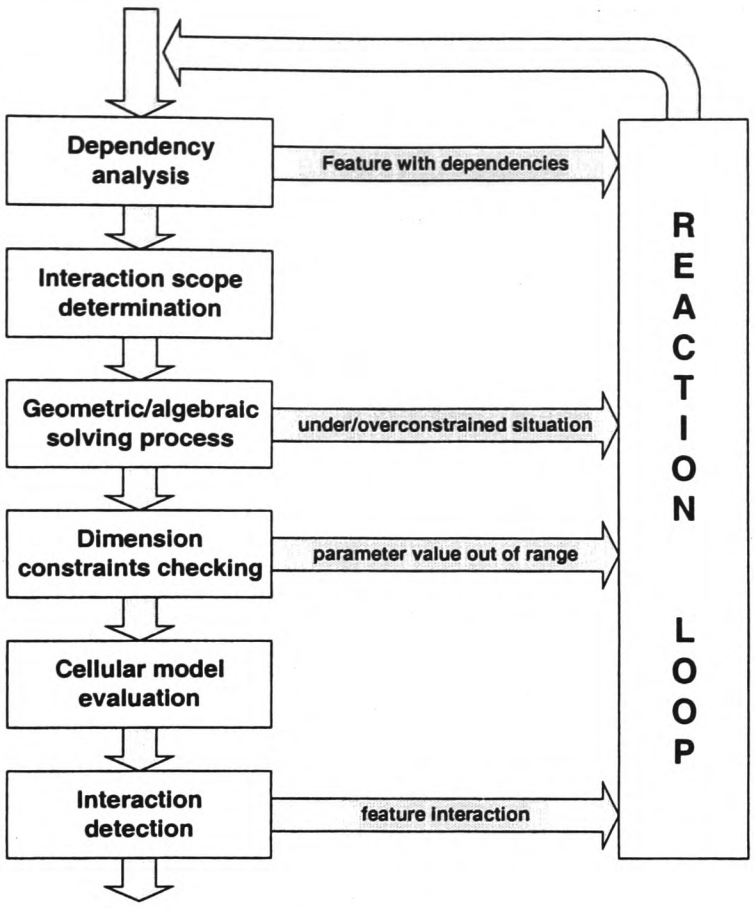


Figure 2-15: Reaction loop

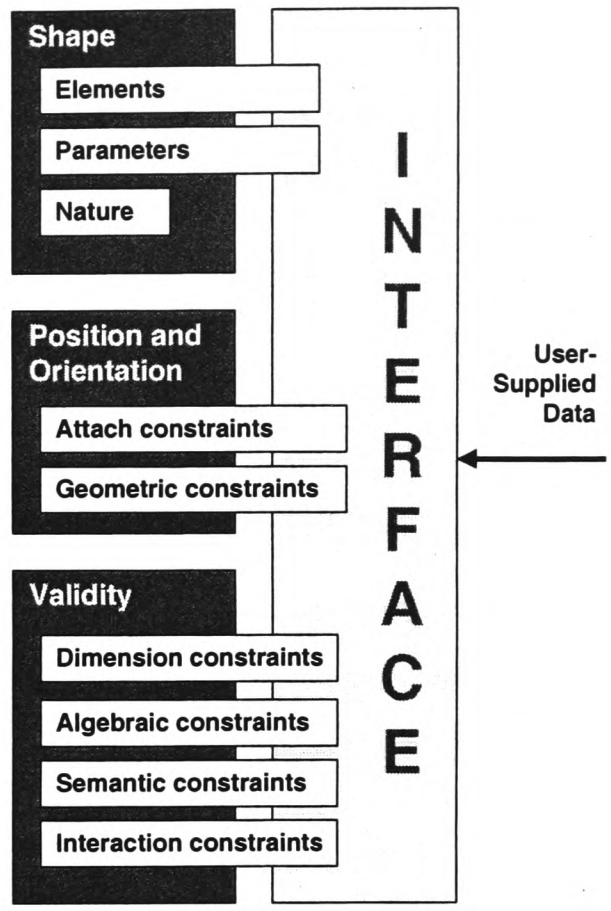


Figure 2-16: Feature structure

Bidarra proposes a complex feature-based system in which the semantic of features is preserved throughout the design process. Features are defined using shape parameters and various constraints through a user interface as shown in Figure 2-16. In particular, semantic and interaction constraints are added in the validity section of features that are used to retain design intent. When the design is modified an automatic validation of the model is performed to detect any undesirable side effects and alert the designer. In Bidarra's own words: "*Feature model validity maintenance is the process of monitoring each modelling operation in order to ensure that all features conform to the semantics specified in their respective classes. Maintaining feature model validity throughout the modelling process requires not only managing all its constraints, but also assessing the conformity of each feature in the model with its validity criteria. This guarantees that all aspects of the designer intent captured in the model are permanently kept*".

Internally, the feature-based system maintains several parallel representations of the model. In particular, it uses an original cellular model **[70]**. The cellular model represents a part's geometry as a connected set of volumetric quasi-disjoint cells, in such a way that each one either lies completely inside a shape extents or completely outside it.

Feature validation is done using the reaction loop pictured in Figure 2-15, which includes several stages:

- **Dependency analysis** ensures that a feature cannot be deleted if others depend on it.
- **Interaction scope determination** reduces workload and guarantees locality of evaluation by finding features potentially affected by a modification.
- **Geometric/algebraic solving process** determines dimensions, position and orientation of all features in the model.
- **Dimension constraint checking** detect out of range dimensions.
- **Cellular model evaluation** updates the cellular model of the part.
- **Interaction detection** detects disallowed interactions in the model. Comparing interactions found inside the updated cellular model with semantic or interaction constraints contained in features does this.

Bidarra's approach provides a very complete and powerful validation mechanism for feature-based models. His use of cellular representations **[70]** for interaction detection allows for detecting a wide range of feature interactions **[56]**. This approach remains a detection only system and does not yet generate redesign suggestion based on the results of validity checking.

2.4.6 Feature mapping

Feature representations are domain (application) dependent. However it is evident that product models must travel through domain boundaries in order to go from design concept to marketable product. Each domain uses different applications to progressively enrich the model and improve its quality. Therefore there is a clear need for applications in different domains to be able to extract only relevant data from the product model and format them according to their domain dependant view. This extraction must take place before any computations or modifications can be performed. In the case of a feature-based model, this process is called feature mapping **[15, 71, 72]**. Feature mapping difference with feature recognition is in the initial state of the model. While feature recognition attempt to generate a feature set directly from a geometric model, feature mapping extract application-specific feature from an existing feature-based model.

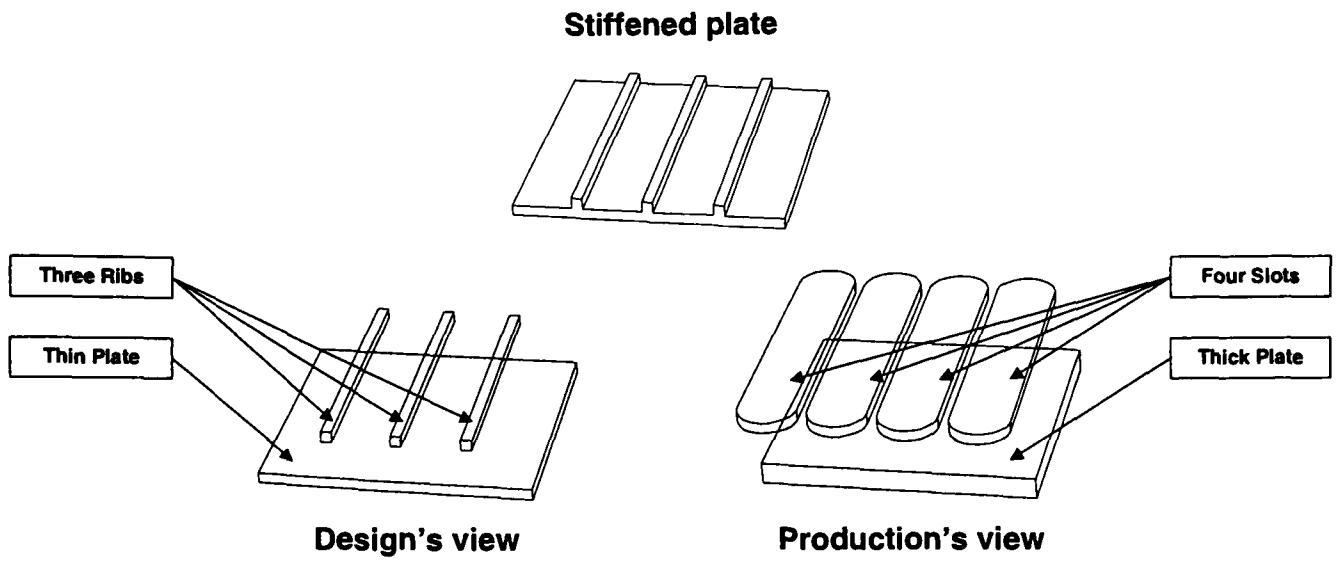


Figure 2-17: Feature mapping between design and manufacture

Mapping between design features and manufacturing features is of particular interest in order to bridge the gap between two critical stages of a product's life. The designer may creates a model using design-specific features. These are features that help him capture his intent inside the model. When the design reaches completion, it must be passed to production engineers for manufacturing. However, the design features used in the model are not meaningful to him. They must be mapped to machining features before he can effectively apply his expertise on the model. For example (see Figure 2-17), a model might contains stiffeners than the designer may adds to the model using ribs. From a production point of view, these ribs will most likely be realised by removing the material between them. In order to integrate the CAD and CAM applications processing this model, a mapping between design-specific ribs and production-specific slots must be performed.

In **[71, 15]**, Shah creates a theoretical framework for understanding feature mapping. He borrows the concept of vector space from linear algebra to classify features. Feature depends upon product type, application and level of abstraction. A combination of these three factors defines a domain designated as *feature space*. Two features spaces of identical dimensions can be *disjoint* or *conjoint*. For spaces with different dimensionality other relations exist. *Projection* transformations allow high-dimensional features to be selectively abstracted to suit lower-dimensional domains. *Conjugate* spaces designate sub-spaces, which contain features that are composed of different variations of the same elements. Figure 2-17 illustrates conjugate relations between three-dimensional elements that are ribs and slots. Associating elements of different sub-spaces creates *adjoint* spaces. Geometric tolerances are a typical example of *adjoint* features.

When two feature spaces are fully disjoint there is no transformation available. For conjoint regions of overlapping spaces an identity transformation is used. Mapping features between conjugate spaces requires geometric reasoning. Common situations that arise in mapping between conjugate features spaces can be classified as follows **[15, 73]**:

- **One-to-one**.

  Features such as hole, slots and pockets might be similarly defined making mapping a trivial operation.

- **One-to-many** (discrete decomposition).

  One feature might be decomposed into several. A counter-bored hole might be realised in two operations. A hole of large diameter might also need several drilling operations.

- **Many-to-one** (discrete aggregation).

  Several features can be aggregated into a single one. A slotted pocket is often given for example. Special tools can be used to drill a succession of hole of decreasing diameter.

- **Many-to-many** (conjugate mapping).

  A group of features can be broken down into primitives and recombined into alternative feature sets. This is the most complex mapping.

- **Specialisation** and **generalisation** mapping.

  Features can be abstracted or specialised to suit the target space. Holes are typically generic entities in design but highly specialised in production depending on dimension ratios and desired quality.

- **Variant re-parameterisation**.

  Feature parameters need to be recalculated. Similar features in different sub-spaces can use different reference systems for positioning and dimensioning.

Feature mapping is vital for integrated CAD/CAM systems. Features are application-dependent. Engineers in different fields use specific features that help them reason on the product model. To achieve true concurrency automated mapping mechanisms should be provided that allows the models to circulate through domain boundaries. Crossing these boundaries should be done without losing the information captured inside the model and with minimum human intervention.

2.5 Mixed approaches

While design by feature offer the best results in terms of capturing additional information inside part models, it lacks the flexibility and ease of use that feature recognition achieves. Several projects have logically tried to integrate the two approaches in order to combine their benefits in a single feature-based system. **[2, 45, 74, 75]** has adopted such mixed approaches.

2.5.1 EXTDesign: incremental feature recognition

Laakko introduces the concept of incremental feature modelling in **[45, 49, 76, 77]** by implementing a hybrid of feature-based design and feature recognition in a single framework. This integration allows the designer to interact with a model using both feature and geometric operations.

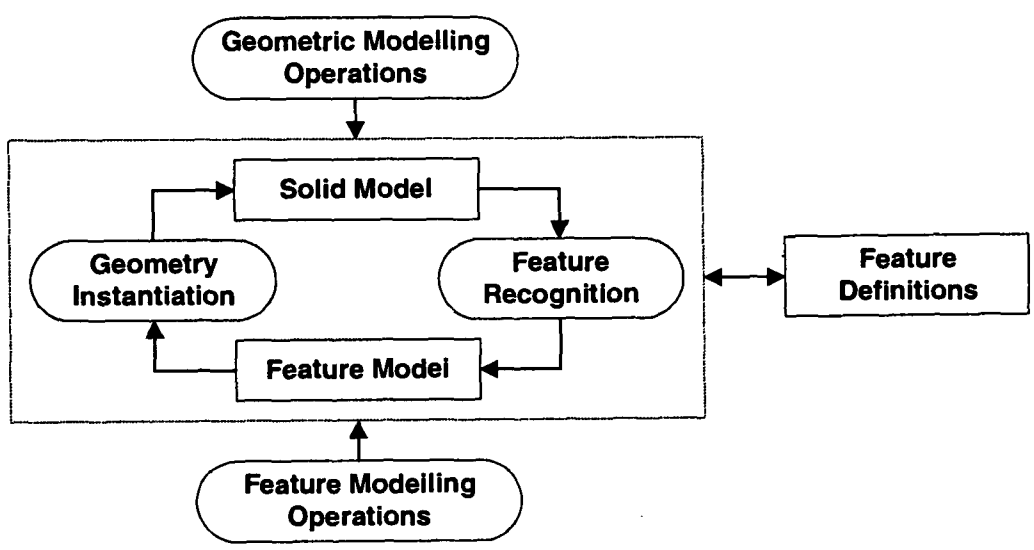


Figure 2-18: Design using incremental feature recognition (from [45])

The EXTDesign system he describes keeps the geometric and feature-based models consistent throughout the design process (see Figure 2-18). Therefore giving the designer the freedom of choosing the most convenient mean of expressing each needed operation. The consistency mechanism uses an innovative incremental feature recogniser capable of detecting changes inside the solid model caused by new or modified feature, while preserving previously recognised feature that remain unchanged.

When the solid model representation is modified (or created), the corresponding feature model representation can be updated by the feature recogniser with the incremental operation. The feature recognition and geometry instantiation processes use information encoded in feature definitions. They allow either representation to be edited while keeping the other representation consistent with the edited one.

Laakko believes "*that the often-mentioned dichotomy of feature recognition and feature-based design is false and misleading, and that a system superior to either can be built by combining the two approaches in a single system*". Using the incremental feature modelling approach, EXTDesign delivers a system allowing editing of both feature and geometric representations.

2.5.2 IF^2: Integrated Incremental Feature Finder

In [20], Han describes **IF^2**, a new system for generating machining feature models of machined parts. This system behaves as a feature recogniser when its input consists exclusively of nominal solid geometry, or as a feature model converter when its input consists of design features. It can accommodate mixed input with both design feature and plain geometry. **IF^2** utilises information available in the nominal solid model of a part, in design features, and in tolerances and attributes.

Unlike Laakko's EXTDesign, **IF^2** aims at generating a feature representation suitable for machining. Although it supports mixed inputs, it only allows manipulations to be performed on the feature representation through creation, deletion or modification. It receives as input a set of currently valid machining features and determines the additional machining features necessary to fully decompose the volume to be removed.

2.6 Conclusion

A definition of feature has been proposed that emphasise the notions of usefulness and context of interpretation. The use of features for mechanical design has been discussed. It has been shown that features extend pure geometric modelling by capturing high level concepts used when reasoning about a product. It was also shown that purely geometric features do not provide an efficient path to integrating design and manufacturing.

The two main approaches to feature-based design, namely feature recognition and design by features have been presented. Feature recognition aims at automatically extracting features from purely geometric representation, thus providing a path from design to manufacture. It is an attractive approach because it accommodates existing geometry-based models. However, it lacks a desirable property, which is the preservation of design intent during design. Design by features requires that model be created from scratch using features. Therefore it is less attractive when large collections of geometry-based design exist. However, feature by design allows models to capture information that is lost in conventional

systems. Designer's intent and meaningful manufacturing attributes can be preserved inside models designed using features.

Feature mapping has also been discussed as a crucial aspect of feature-based design. Features usually need to be application-specific to offer their full benefit to users. Therefore, a mapping mechanism must be provided that can transform feature representations from one application to another if strong integration is to be achieved.

Finally, hybrid systems using both feature recognition and design by feature were presented. Such mixed system accommodating both feature and geometric representations show the road ahead in fully integrated systems for design and manufacture.

Chapter 3

Design and Manufacturability

3.1 Introduction

This chapter presents the concepts relating to manufacturability analysis of feature-based models and in particular design for CNC of 2½D manufacturability (DFM). It concentrates on the domain of machined mechanical components.

An introduction to the concepts of manufacturability is given in the first section. The main section discusses the core concepts involved in measuring the manufacturability of machined mechanical parts. It includes a discussion, on various levels, of assessment that can be achieved and a review of different criteria that can be used. The particular process limitations of machined components are discussed in detail. A survey of manufacturability analysis systems is presented before a summary of conclusions on the matter.

3.2 Background

The competitive market place that confronts today's industry is a very strong drive for innovation. Products must be developed quickly, produced cheaply and achieve quality levels that are constantly rising. In order to tackle this daunting challenge manufacturing industry is seeking tighter integration of the information systems used along the different stages of a product's life. This integration is seen as the main means to achieve higher concurrency during product development. In particular, the traditional "over the wall" communication between design and manufacturing is no longer acceptable. Indeed, it is

recognised that the transition between design and manufacture is a key area in which tight integration could bring major benefits.

The traditional approach creates a dichotomy between designers and machinists. On the one hand, the designer's job is to capture necessary functionality inside a part model according to the desired specifications. On the other hand, the machinist must assess the feasibility of parts according the available processes. He must also generate production plans that will optimise machining cost as well as flow of material and parts. No real communication exists between the two activities. Of course the designer might have some knowledge of the machining process but probably not enough to fully assess the manufacturability of designed parts.

A strong integration between CAD and CAM systems is a major means of concurrent engineering. It makes it possible to shorten the development cycle by bringing downstream concerns up-front to avoid costly re-design iterations at the manufacturing stage. Indeed Gupta notes that "*one of the primary goals of concurrent engineering is to build an intelligent CAD system by embedding manufacturing related information into CAD systems. In an intelligent CAD system, DFM is achieved by performing automated manufacturability analysis- a process which involves analysing the design for potential manufacturability problems and assessing its manufacturing cost*" **[12]**.

3.3 Measure of manufacturability

A DFM implementation requires means of measuring manufacturability, which is assessing parts in regards to manufacturing issues. The assessment can be done with different levels of accuracy and requires that manufacturability criteria be defined.

3.3.1 Assessment level

There are many different levels of assessment that can be used for measuring manufacturability.

- **Binary**:

  The most basic manufacturability assessment will yield a PASS or FAIL result according to the chosen criteria. A part that barely achieves the required properties produces the same result than a fully optimised design. Similarly, a part containing a single minor problem cannot be distinguished from a massively flawed one.

- **Qualitative**:

  Here, analysis generates a more subtle assessment of parts. Designs are given qualitative grades (such as 'poor', 'average' and 'good') based on their manufacturability by a certain production process. Although an improvement over binary level, qualitative ranking can be difficult to interpret. In **[12]**, Gupta points out that when several analysis systems are used (for example machining and assembly), such qualitative results are difficult to aggregate into a meaningful global assessment of a part.

- **Quantitative**:

  In quantitative schemes, numerical values are applied to parts instead of qualitative attributes. However, such results might still be subject to interpretation. Partial values for specific domains are also difficult to combine into useful global results. However, quantitative values applied at feature level **[58]** can provide valuable feedback to designers by pointing out problematic areas in assessed parts

3.3.2 Manufacturability criteria

Assessing manufacturability of a design is done according to production-specific criteria chosen according to the processes envisaged. In **[78]**, Ong proposes a manufacturability hierarchy that divides criteria into two main categories, namely, geometrical criteria and technological criteria (see Figure 3-1). A third level in the hierarchy offers fine-grained manufacturability attributes.

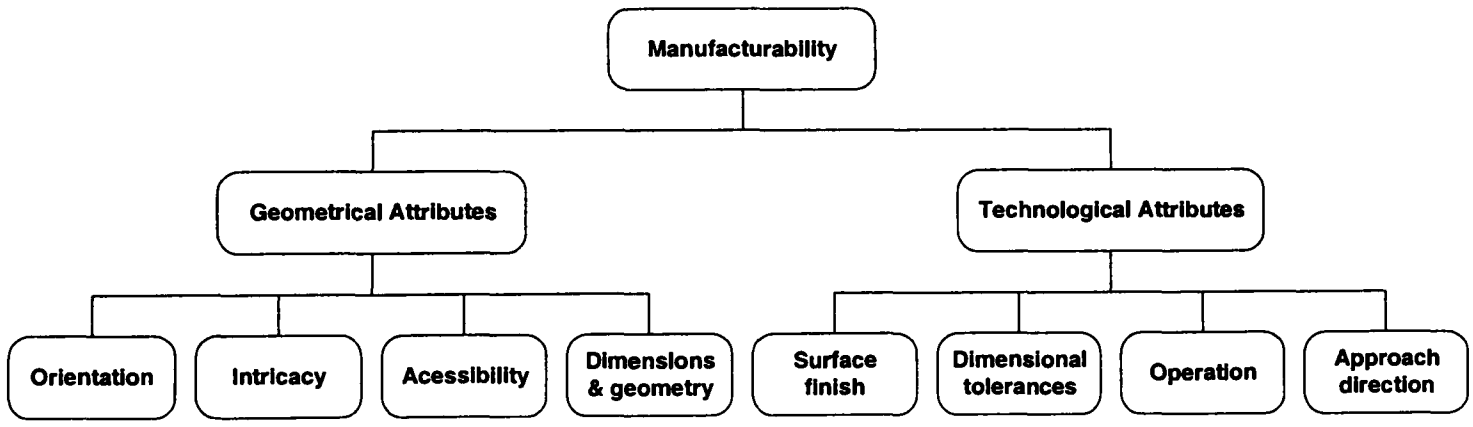


Figure 3-1: Manufacturability hierarchy (from [78])

The attributes defined by Ong are as follows:

- **Orientation**: This is a formalised attribute from the observation that faces that are parallel, or perpendicular, to the principal axis of 3-axis milling machines are easier to create that surfaces that are inclined.

- **Intricacy**: Defines the amount of details or complexity in a feature or model. A rough estimate of intricacy can be obtained by counting the faces of a feature or model. Higher complexity usually translates into higher difficulty during machining **[78, 79, 80]**.

- **Accessibility**: The surfaces to be machined must be easily accessible so as not to need the construction of a special tool.

- **Dimension & Geometry**: Cutting tools have standard dimensions and geometry that can affect the shape of features to be machined. Tools must withstand high forces during machining, therefore their dimension ratios are limited to offer sufficient strength. Features aspect ratios should also stay within values obtainable using these geometrically "limited" tools. Also standard dimensions in features can greatly affect the difficulty of their manufacture. For example, if a pocket possesses a corner radius equal to a standard tool diameter, it is easier to machine than one with a non-standard corner radius.

- **Surface finish**: Machining cost increases with finer finish, as additional operations are required to obtain them.

- **Dimensional tolerance**: Stringent tolerance specification will increase the difficulty and cost of production. Machining centres have physical limitations in terms of position accuracy that limit achievable tolerances. Vibrations and heat-induced deformation are other limits that must be taken into account.

- **Operation**: Each type of machining operations has its own capabilities concerning dimensions, tolerances and finish **[18, 19]**. When machining a feature, feasible operation have to be selected based on their capabilities.

- **Approach direction**: A rule-of-thumb is for the tool to preferably approach a feature using the access direction with the smallest depth **[78]**. When several access directions are available, it is possible to rate them according to that rule.

These attributes cover most aspects of manufacturability analysis and provide a good basis for discussing manufacturability criteria further. A detailed presentation of problems relating to the machining process follows.

3.3.2.a Process limitations

Mechanical parts are usually designed with prior knowledge of the manufacturing processes that will be used for its production. Therefore designers should attempt to take into account the limitations of these intended production methods. For example, a part for casting should contain blended corners rather than sharp angles. Also, milled components overall dimensions are constrained by machine tool dimensions. Specific criteria for the cutting process involved in producing 2½D milled components are presented below.

Process limitations: ① **Tool access**

In order to be machined, a feature on a part must be accessible to the cutting tool. This access implies that the rotating tool and spindle must be able to travel along the necessary trajectory that will generate the desired removal of material without interfering with the finished part or other obstacles such as clamping devices. Access to a feature can be *direct* when the feature emerges at the surface of the blank or *indirect* when it is granted through another machined feature. Indirect accessibility of machining features translates easily into temporal-anteriority rules that can be used as a basis for machining operation sequencing.

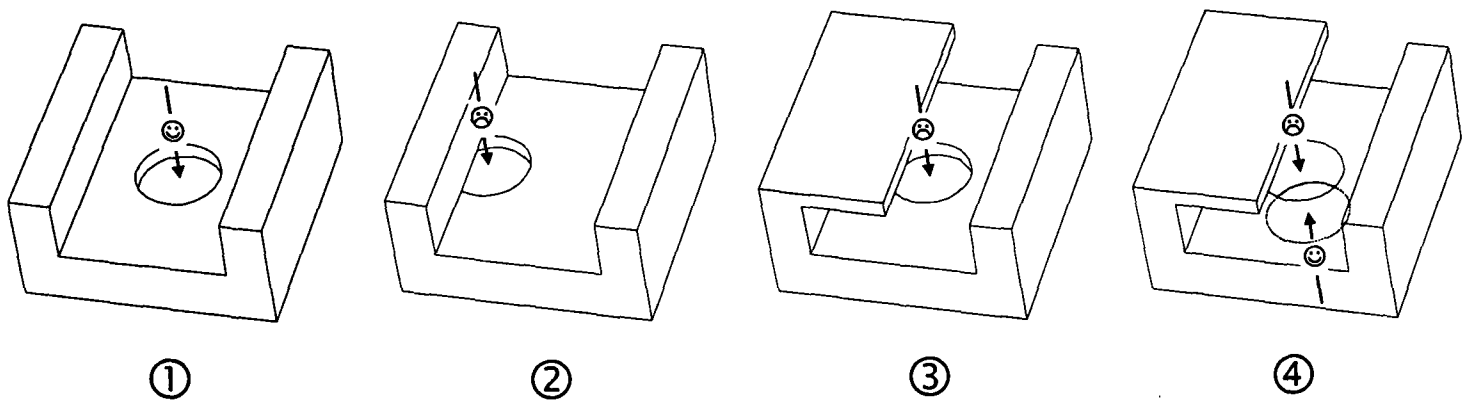


Figure 3-2: Tool access configurations

Figure 3-2 illustrates some access configurations concerning a hole on a simple component. Figure 3-2① shows the hole sitting at the bottom of an open slot and can be accessed by the cutting tool. Note that this access is indirect and the feature is only accessible *after* the slot has been created. In Figure 3-2② however, the same hole has been offset to the side, making part of its entry face directly obstructed by the block. The hole in Figure 3-2② is not machinable in this configuration because no ordinary tool can obtain access to machine it. Figure 3-2③ shows another access obstruction. In this case, the entry face lies completely open at the bottom of a slot but tool access obstruction is still committed by the block. Figure 3-2④ is a modification of Figure 3-2③ where the hole is made through, thus creating a

possible access route through the back face of the part. Note that, unlike the access in Figure 3-2①, the access through the back-face does not require the slot to be machined first.

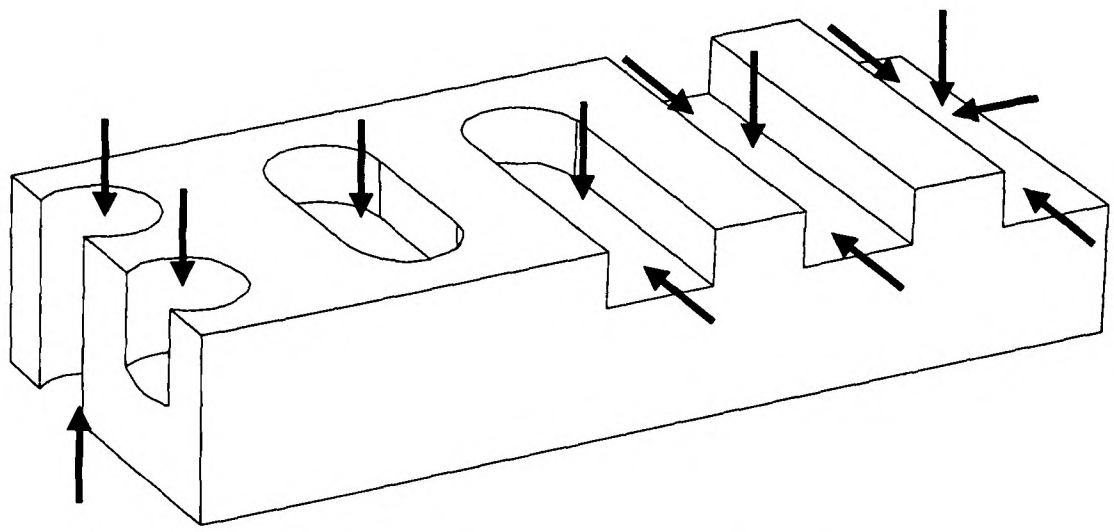


Figure 3-3: potential access directions

In **[62]**, Salmon reviews different techniques for detecting access problems. Most approaches use embedded knowledge about features tested. In particular, each feature type possesses one or more potential access direction that can be tested for obstructions. Figure 3-3 gives a few examples of access directions for various features. A normal hole has one, a through hole two. A slot possesses different number of access directions depending on its type. A blind slot has one, a half-through slot two, a through slot three. The right-most feature in the demonstration part is a step (but might be created using a slot in manufacturing). It has 4 possible access directions.

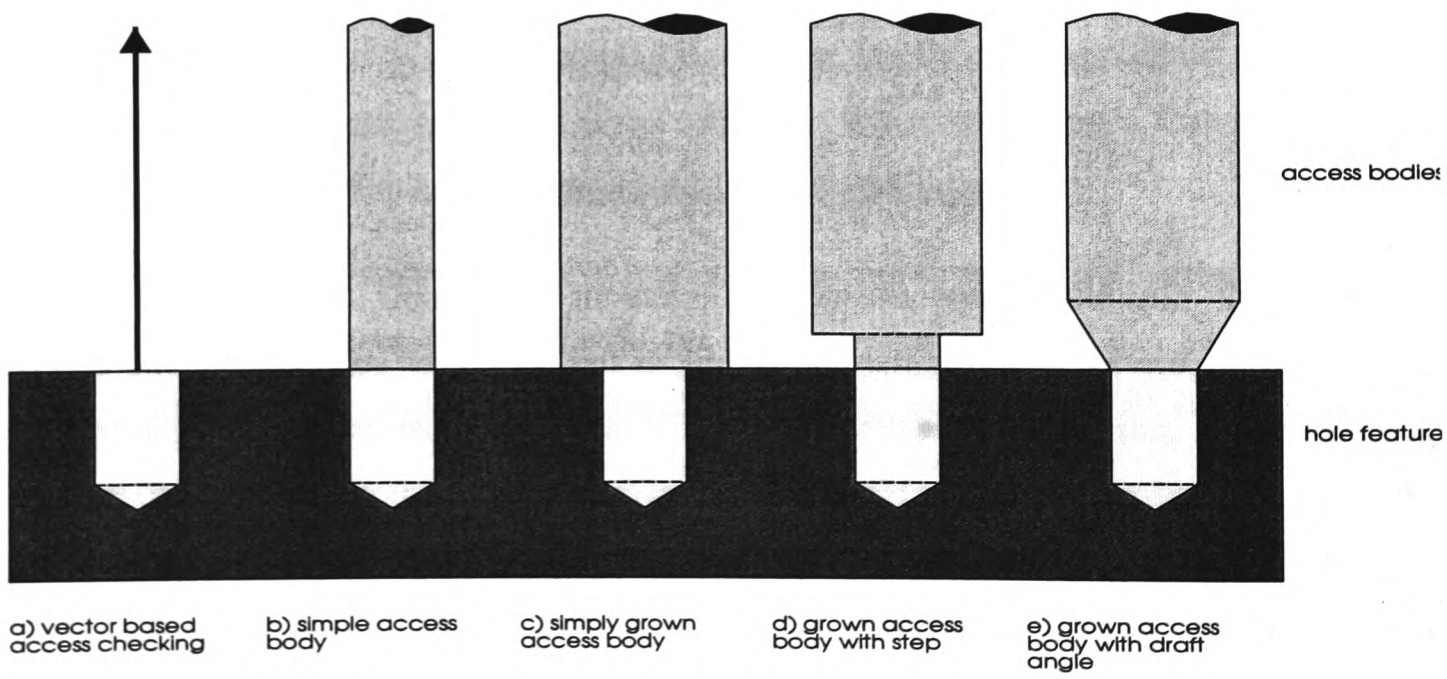


Figure 3-4: Access body types (from [62])

To determine the accessibility of a feature, manufacturability analysis systems test possible access directions for obstructions. Most techniques generate a geometric entity (referred to as an access body) emerging from the feature and following the access direction. The access body can be tested against the blank, other machining features in the model and the desired finished geometry. Intersection with the finished part means the access direction is obstructed and can not be used for machining the feature. Intersection with the blank indicates that direct access is not provided and indirect access must be found for the part to be manufacturable. Intersection with other machining features can indicates indirect access (though not always) and can be used to infer temporal precedence relationships between features. Different types of access bodies can be used. Simple vectors can detect some problems but are not reliable. More complex bodies can be generated to better represent the volumetric properties of tools and spindles (see Figure 3-4). Ultimately, an access body used for testing manufacturability should be retrieved from the production site and reflect the real geometry of available tools and tool holders.

Accessibility checking is of critical importance when analysing manufacturability. Failing to detect tool obstruction in a fully automated CAM system could result in grave damage to tools, parts and even machine tools. Also, access checking can help generate precedence relations among feature useful for process planning.

Process limitations: ② **Part handling, Fixturing**

Fixturing is critical in manufacturing. Without fixtures, there is no machining possible. Wright expresses the importance of fixtures as follows. "*To peel an apple, a human being clamps the fruit in one hand and the knife in the other, The interaction forces are low, and accuracy matters little. [...] To face-mill a large block of steel with an 8-inch carbide cutter [...] the experienced machinist at first pays considerable attention to clamping the part on the milling machine's bed. The machinist know that the interaction forces will be several hundred pounds and that accuracy are being specified to a few thousands of an inch*" **[81]**.

Fixtures are used to constrain workpieces during machining processes. Fixtures locate and hold the workpiece in position and ensure that it is in a state of equilibrium, and that dimensional accuracy is maintained throughout the manufacturing operation **[82]**. The aims of fixture design for prismatic component is to devise a configuration of clamps and locators that:

- position a workpiece so as to provide required tool access to machining features

- guarantee this position and orientation even when submitted to cutting forces
- ensure that clamps and locators do not interfere with the cutting tool trajectories

Positioning and holding is achieved through physical contact between the workpiece and the clamping devices Therefore, successful fixture design depends heavily on the actual geometry of the workpiece and requires complex geometric reasoning on the component 3D representation. While fixturing for a single-face strictly 2½D part is relatively straightforward, multi-sided components present a challenge.

Because fixturing is so critical to the successful production of machined parts, manufacturability analysis should provide evaluation of the fixturability of components. Research work on automated fixture design in feature-based system **[82]** provides a glimpse of what such an evaluation entails.

Process limitations: ③ Fe a ture aspect ratio

Cutting tools used for machining are subject to very high level of stress. During machining, huge forces are applied by the tool on the part and therefore by the part on the tool. Tools must be structurally strong in order to minimise flexion that would compromise achievable tolerances. Tools that cannot withstand the cutting stresses are useless as they damage the part and themselves during machining. This is different from normal tool wear that involves degradation of the cutting edges of the tool rather that its overall structure.

Therefore, physical limitations exist in tool dimensions and dimension ratios. In particular, rotating cutters are limited in their length/diameter ratio. Indeed this ratio is critical for both drills and mills and cannot exceed limit values. During entry in the material a very long drill does not benefit from the lateral support of the hole and might bend. Similarly, a mill of small diameter might bend during side milling. Logically, these limitations in the aspect ratio of cutting tools are transmitted to features machined using them. For example, because a mill of small diameter can not be very long, narrow slots can only be "easily" manufactured with limited depth. Also, very deep holes of small diameter are expensive to create because they require multiple operations using drills of increasing length (to avoid bending).

Feature with problematic form aspects should therefore be detected during analysis to guarantee optimal manufacturability of designed components. Detection of these situations is not necessarily difficult but requires knowledge about the process capabilities.

Process limitations: ④ **Machining Forces and Part Integrity**

Mechanical machining using cutting tools can generates very high stress on both the tool and the part being machined. According to Chang, "*in orthogonal cutting, the resultant force, F_r, applied to the chip by the tool lies in a plane normal to the tool cutting edge (see Figure 3-5). F_c is the major cutting force and F_t is the thrust force. The cutting force can be expressed roughly as a product of the specific cutting resistance, k_s, and the cross-sectional area of undeformed chip*" **[83]**.

$$F_c = w\, h_c\, k_s$$

$$F_t = b\, w\, h_c\, k_s$$

Where F_c, F_t = cutting forces

b = coefficient empirically determined by tool geometry

w = width of undeformed chip

h_c = thickness of undeformed chip

k_s = specific cutting resistance

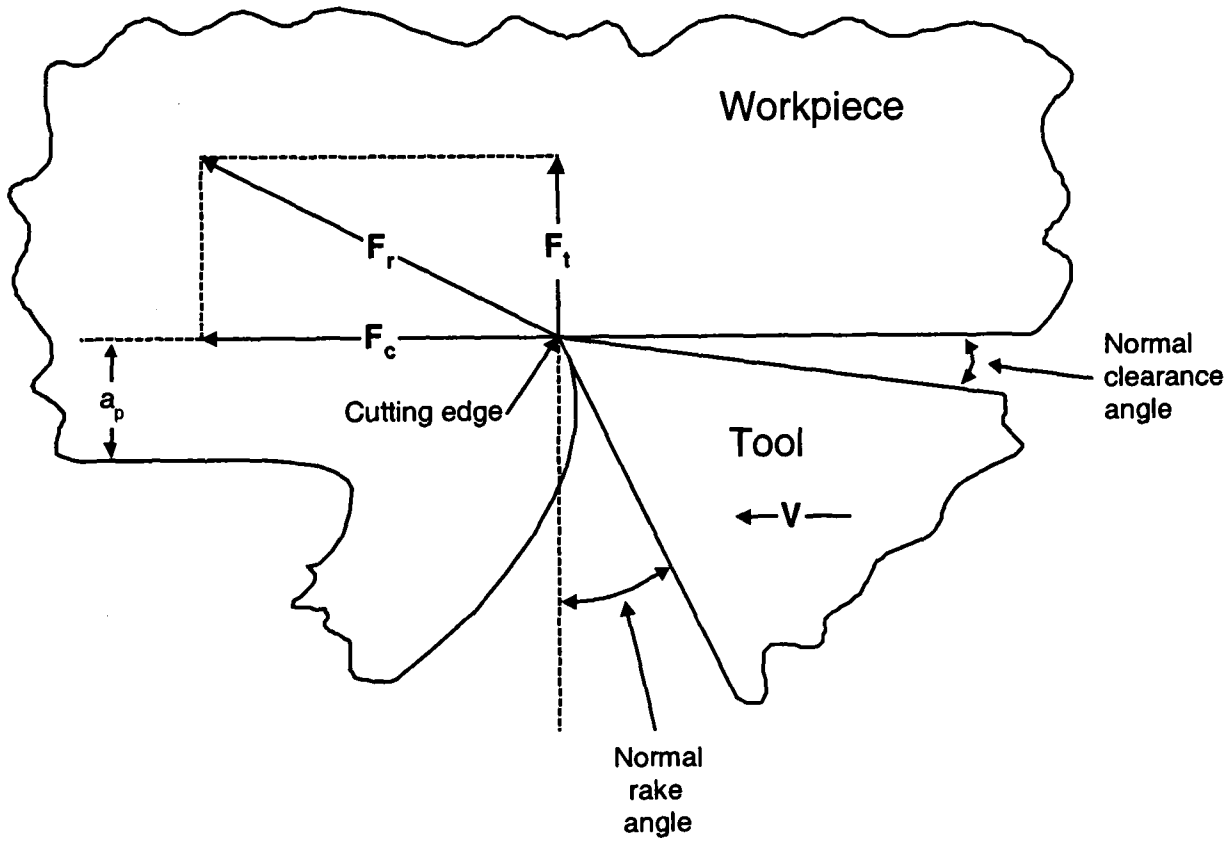


Figure 3-5: Cutting forces geometry ("merchant model" in [83])

In some cases, these forces might be too high for a given part and compromise the part geometry. A typical example is that of thin wall between features that might bend or rupture during machining. In cases where the thickness of the workpiece is small, the thrust force, F_t,

applied during cutting can become problematic. When the workpiece become too thin, it can no longer withstand the forces applied by the cutter and flexion occur (see Figure 3-6). Thinner still and the wall can rupture during machining. Of course, cutting parameters can be modified to minimise forces. F_t is a function of both chip thickness and chip width, which are determined by the feed and the depth of cut respectively. Using small feed and depth of cut can therefore reduce the thrust force at the expense of longer machining times. However, minimum values for these also exist and no traditional cutting process can realise arbitrarily thin walls. Furthermore, additional factors such as vibrations and heat-induced deformation also contribute to render thin wall manufacturing difficult using traditional cutting methods.

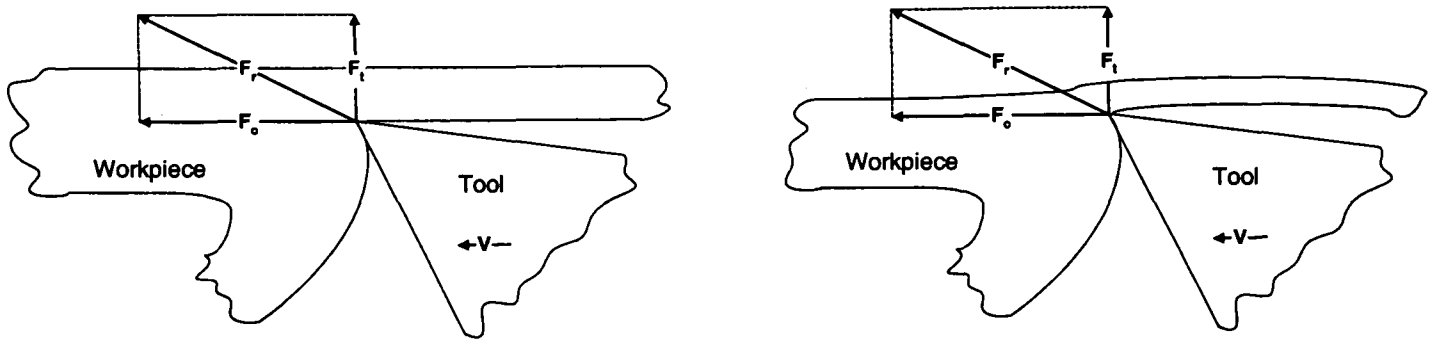


Figure 3-6: Thin wall deflection during machining

Various techniques are used to manufacture thin walls. Special fixturing techniques can offer additional support to the workpiece during machining **[84]** at a price of lost accuracy. High velocity machining is also used for thin walls in the aerospace industry as it greatly reduces cutting forces **[85]**.

For most manufacturing industries, conventional machining techniques are still the norm. Therefore, such critical conditions must be detected to ensure the final quality of produced parts. In particular, even minor deflections during machining can dramatically alter geometric tolerances achieved and must be avoided.

Detecting thin walls is mostly a matter of evaluating proximity between features inside part models. An acceptable proximity distance is chosen for a component according to the material and the machining centre used. This limit value represents the smallest dimension achievable on a component. It is compared with thin parts of the design to determine their manufacturability. For example, Gupta searches thin walls by faceting the component and evaluating the separation distance between close, non-adjacent faces **[86]**. This evaluation removes the difficulty of handling curved surfaces but is dependent upon the quality of the faceted representation. Salmon uses the Minkowski sum to "grow" features by the proximity distance and performs Boolean operations between the grown features to detect thin walls

[62]. Two non-intersecting features whose grown bodies intersect are marked as creating a thin wall problem. This approach is more rational and is not subject to representation accuracy. However, it requires some tweaking to allow detection between a machining feature and the outside world.

Process limitations: ⑤ **Ob tainable quality**

One major concern in modern manufacturing is to ensure total quality of produced parts. All parts produced should be so within the specified geometric tolerances. For these reasons, manufacturability analysis should offer automated validation of tolerances and quality attributes of parts. Tolerance analysis in terms of production capabilities is highly useful for process planning as it helps eliminate problematic configurations before going to production.

Some attributes such as form tolerances and surface finishes are easily validated after comparing them with the known maximum capabilities of potential machining processes. Intrinsic tolerances (see section 2.4.3.a) of features can also be validated simply because a feature will usually be machined on a single machine, using a single setup. Tolerances and finishes specified in a design might simply be too high for the intended production processes and must be revised to accommodate manufacturing capabilities.

Extrinsic tolerances are more difficult to analyse as they involve several features that potentially require to be machined in different setups. Typical examples of situations were tolerances cannot be achieved are given in **[86]**. On a 3-axis milling machine, the number of setups required to produce a part is mostly dictated by tool access. If all features can be accessed through an identical access direction, a single setup is required (providing fixturing is possible) and even tight tolerances can be achieved. Otherwise, several setups are needed. In that case, tolerances between features machined in different setups should be loose enough to accommodate setup changes.

3.3.2.b Time and Cost

Gupta argues that "*in general, a design's manufacturability is a measure of the effort required to manufacture the part according to the design specifications. Since all manufacturing operations have measurable time and cost, these can be used as an underlying basis to form a suitable manufacturability rating.[...]Moreover they represent a realistic view of the difficulty in manufacturing a proposed design and can be used to aid management in making make-or-buy decisions*" **[12]**.

Total quality and the importance of producing mechanical parts within specified tolerances have just been discussed. Quality specification of mechanical parts should therefore be validated during manufacturability analysis. However, a design cannot be over-specified so as to include tight dimensional tolerances **[83]**. The surface finish, for instance, should be specified to be the largest value possible that meets the functional needs. The impact of over-specified design on a company's profitability should not be understated. Specifying parts requiring secondary and tertiary operations when a single operation alternative exist can greatly increase both time and cost. Some specific industrial sectors, such as aerospace and armament, can "afford" to produce over-specified parts. However, in a market-oriented industry, reduction of production cost is paramount to increase profitability.

Production time and cost are second only to the physical feasibility of parts. In fact, most industries consider them prime factors for selecting between alternate design or production plans. Cost in particular is a driving force in re-design and optimisation of existing parts.

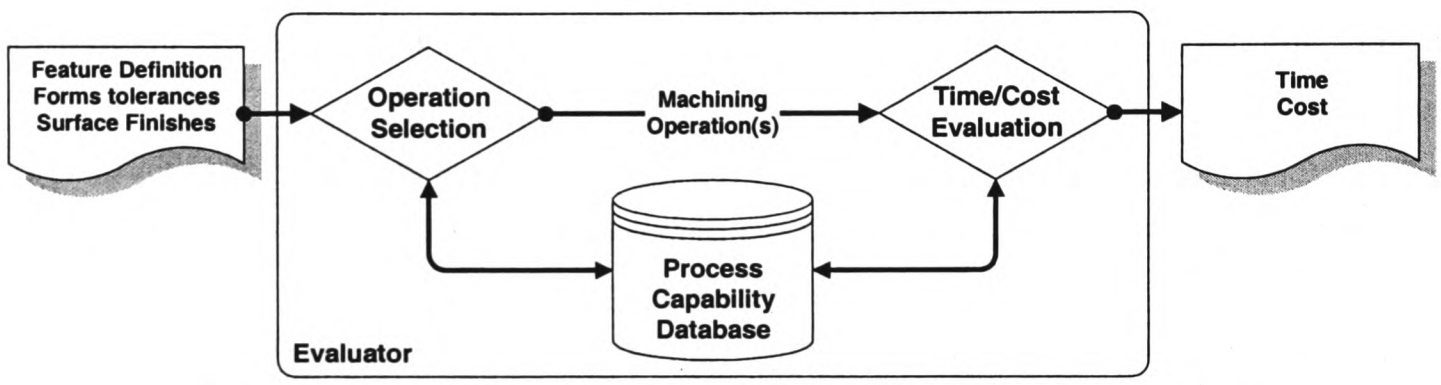


Figure 3-7: Time/Cost evaluation using process capability database

Knowledge about process capabilities, operation time and cost is needed. Evaluation of time and cost for each individual feature in a design can be done as shown in Figure 3-7. Feature definitions, intrinsic tolerances and surface finishes are fed into the evaluator. The evaluator contains a database containing models of the process capability, as described in **[18]**, with additional cost and time values for each machining micro-cycles. The database is queried to find appropriate machining operations. That is to say operations that permit the realisation of the defined features within the specified tolerances. The database is queried again for estimated time and cost for selected operations. The individual results obtained from the database can be aggregated into a global estimated time and cost to machine a feature or an entire component.

Gupta's analysis system, the Interactive Manufacturability Analysis and Critiquing System (IMACS), estimate machining time for each operation using a simple function:

$$T(o) = L \times \frac{n}{f}$$

L = Estimated length of tool trajectory
n = Number of passes
f = Feed

For each type of machining operation, the trajectory length is estimated by the parameters of its machining feature. For simple feature estimation this is straightforward, for complex features, generation of the cutter path might be necessary. IMACS also provide estimates of setup time for parts needing several setups.

3.3.2.c Other considerations

Other considerations can also be taken into account when evaluating manufacturability of machined parts. This thesis is principally concerned with analysing components from a machining point of view to ensure part feasibility and the existence of potential process plans. However, other aspects also influence design choices.

Design for **assembly** is used to bring the concerns of assembly up-front during the design stage. Such systems analyse multi-part components and assess the difficulty of realising the assembly. Specific criteria such as parallelism, assemblability and redundancy are used to assess designs.

Environmental issues are a growing concern across all industries. The manufacturing industry is no exception. Governmental regulations and a growing preference for "green" products push manufacturer to introduce new manufacturability criteria such as comparative waste stream assessment **[87]**. Analysing the environmental impact of dissimilar waste is a complex problem in which different factors (toxicity, flammability, or reactivity) must be considered. Reduction of energy consumption during machining is a straightforward economic factor but can also be considered an environmental issue.

3.4 Level of automation

The level of automation of analysis systems concerned with manufacturability varies between systems.

- **No automation**:

  This is the approach adopted by most existing systems. Designs are evaluated for manufacturability and results are provided to designers to analyse and act upon.

Despite the lack of automation, it can be an efficient method if the assessment level is detailed enough and geometrically localised. In particular, analysis yielding quantitative values for each feature in a model can help pinpointing problematic areas in a design on which designers should concentrate to obtain viable (from a production point of view) solutions.

- **Redesign suggestions**:

  It might not be enough for a manufacturability analysis system to detect manufacturing problems inside designed parts. Even detailed results, listings of potential production impasses for well-defined areas in a design might be wasted. Indeed, designers might not have the specialist knowledge in manufacturing necessary to solve these problems. Automatic generation of re-design suggestions proposes to volunteer possible solutions for problems detecting during analysis **[88]**. This is a very complex problem however. For example, it is difficult to evaluate the consequences of proposed modification in terms of functionality of parts. Also, while generations of suggestion for a single feature is relatively simple, it becomes very difficult for interacting features.

- **Automatic corrections**:

  Here, the automated manufacturability analysis system goes even further and automatically corrects some (or all) of the detected problems in designs. This might be considered undesirable and even dangerous in light of the difficulty predicting the consequences of such modifications in terms of functionality. However, although fully automated correction might be detrimental, partial automation could lighten workload by taking care of non-critical and well understood problems. For example, the automatic removal of erroneously tight tolerances and surface finishes might be a good candidate for automatic correction during manufacturability analysis.

3.5 Survey of existing systems

In [12], Gupta provides a comprehensive survey of automated manufacturability analysis. This survey covers all manufacturing areas, including machining, assembly, printed circuit board (PCB) production, casting and others. This section concentrates on manufacturability analysis systems for machined prismatic parts.

Cutkosky and Tenenbaum developed a feature-based design system for machined mechanical parts named *Next-Cut* **[89]** (successor of *First-Cut* **[90]**) that warns designers of potential manufacturability issues. It allows designers to create mechanical components by subtractive synthesis using blanks and machining features. Next-cut attempts to maintain consistency between concurrent views of the designed product reflecting both design and manufacturing. Maintaining consistency involves explicit and implicit dependencies. Explicit dependencies in which feature and machining operation can be linked are addressed through procedural propagation of parameters. Implicit dependencies are harder to track and are addressed by simulations. A process plan is generated, simulated and the result analysed for potential problems. Gupta argues that "*Next-Cut requires that the designer has good knowledge about machining processes in order to select the most appropriate feature set for machining. Failure to do so may produce incorrect analysis*" **[12]**.

IMACS, the **I**nteractive **M**anufacturability **A**nalysis and **C**ritiquing **S**ystem, is an interdisciplinary project developed at the University of Maryland. The ultimate goal of the IMACS project is to provide tools for manufacturability analysis as part of the CAD systems used by designers, thus reducing the need for redesign and decreasing lead time and cost. Concepts from different research work are implemented within IMACS. One of the fundamental operating principles of IMACS is to systematically generate and evaluate alternative operation plans for machined parts **[91]**. Automatic feature recognition is used to extract alternative feature-based models (FBM), which are evaluated for manufacturability **[92, 93]**. Failure to generate a possible plan indicates parts that cannot be machined with the given set of machining operations. When successful plans are found a manufacturability rating is calculated that reflects machining time of parts. IMACS provides feedback concerning unmachinable parts by detecting shape-related problems as well as dimension and tolerance related problems. Upon detection of such problems, generation of redesign suggestions can be achieved **[88]**. Alternative FBM containing combinations of altered machining features are generated and proposed to the designer.

FROOM, the **F**eature **R**elation used in **O**bject **O**riented **M**odelling, is a prototype computer system for the (re)design of mechanical products. It was developed at the University of Twente (Netherlands) and is presented in Salomons Ph.D. thesis **[94]**. FROOM is a redesign support tool aimed at bridging the gap between design and manufacture through the use of features. FROOM uses an interactive feature definition process to identify external and internal constraints that describe features **[52]**. It also supports assembly constraints, variational constraints and parameter constraints. The resolution of these different constraints

is used as a basis for collaborative design between CAD and CAPP. Conflicts in expressed constraints indicate potential problems and are used to trigger communication between collaborating product developers.

In **[62]**, Salmon presents **FODDS** 2 (**F**eature **O**riented **D**etailed **D**esign **S**ystem) which provides a feature-based design environment (design by feature) capable of testing manufacturability of 2½D parts. FODDS2 focuses on geometric reasoning for process planning. Salmon argues that successful process planning requires generation of anteriority constraints between features and presents inferencing algorithms to detect anteriority errors.

The **SPIFF** prototype system, developed at the university of Delft (Netherlands), is a multiple-view, feature-based modelling system, whose goal is to experiment with new techniques on constraint management, view conversion, and model validity maintenance. These techniques can support product development in all its phases, from early conceptual design to final manufacturing planning. Novel concepts researched by Bidarra, Bronsvoorst, Noort, and others are implemented inside SPIFF. This includes functionality such as feature validation, multiple view maintenance and feature interaction management **[56, 63, 64]**. The feature validation aspect is of particular interest because it allows verifying features against manufacturing criteria during design stage.

3.6 Conclusion

The motivations for manufacturability analysis have been presented. Increasing competition requires shortened time to market, higher product quality and reduced production costs. Manufacturability analysis brings manufacturability concerns up-front during design, helping design decisions, which account for a large proportion of later costs. Two generic aspects of manufacturability analysis were discussed. Namely, level of assessment and level of automation. Levels of assessment represent the granularity of results returned after analysis. They can range from a component-wide binary PASS/FAIL to specific quantitative values affected to each feature. Levels of automations describe how users might interact with the analysis system. While most existing approaches provides limited automation, some can generate re-design suggestions to the designer.

The specific manufacturability criteria used for machining were reviewed. In particular, problems relating to tool access, machining forces and obtained quality need to be addressed during analysis in order to guarantee the physical feasibility of a part. Time and cost

evaluation is also critical in terms of usefulness for most manufacturing industries, where cost is at least as important as functionality.

Manufacturability analysis of parts during design is a step towards concurrent engineering. It allows manufacturing problems to be detected and dealt with at early design stages where modifications are relatively inexpensive. It represents a partial integration of CAD and CAM, and a major improvement over to the traditional "over the wall" communication between designers and machinists.

Chapter 4

Agent Technology

4.1 Introduction

This chapter introduces the concepts related to the widely used terms "agent" and "multiagent systems". A definition of agency is discussed and an agent checklist is proposed that suit this thesis' approach. Each item in the proposed checklist is then explained in more detail and analysed. Several classifications of agents are proposed which are based on important aspects. Particular attention is paid to critical issues such as agent communication, co-ordination and emergent behaviour. This chapter also provides a description and discussion of the different technologies already developed to make multiagent systems a reality.

4.2 What is an Agent?

Autonomous agents, software agents and multiagent systems are all part of a currently very popular area of research in artificial intelligence, robotics and computer science. The concept itself however is not new since Nwana **[95]** traces the origin back to 1977. It has however enjoyed a resurgence of interest since 1990 due to the rise of distributed computing and the rapid evolution of computing technologies that makes parallel computing an affordable possibility. However, victim of a fast adoption in very different domains, consequently the term agent is quickly losing a meaning that was once recognised and supported by the research community. Indeed, Hendler deplores that, these days, many aspects of agents have confused terminologies and multiple threads that all blend together

[6]. It is therefore, the aim of the following sections to present as clear a picture as possible of agent technologies.

4.2.1 A possible formal definition

The quest for a universally accepted formal definition of the concept of agency is still on but already looks like a lost cause. Indeed it seems that each new research or application claiming to be "agent" flavoured tries to come up with its own definition. Therefore adding to the existing confusion over this sensitive issue. However, following a very interesting survey of the significant definitions of agency, Franklin proposes its own *formal* definition of autonomous agents:

"*An **autonomous agent** is a system situated within and part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to effect what it senses in the future.*" **[96]**

It is a worthy effort in that it tries to narrow the field of potential agents without being too restrictive in doing so. It is successful in capturing several key agency concepts such as autonomy, time continuity and environment perception without being too lengthy and complex. However, and by Franklin's own admission, the applicability of such "formal" definitions becomes doubtful in extreme cases. For instance, this definition applies equally well to a human being and to a simple thermostat!

The lack of a universally accepted definition of *agency* is mostly due to the astonishingly diverse range of domains taking interest in it. Biological simulations known as artificial life, data gathering on the Internet, smart computer-user interfaces, distributed computing, Turing tests and autonomous robots are just a few examples of the variety of fields covered by agents. It is no surprise that such diverse domains fail to agree on what defines an agent.

4.2.2 Agency check list

Rather than entering the fray and proposing yet another conflicting definition of agency the choice was made to take the approach adopted many **[96, 97, 98, 99, 100, 101]** and propose a list of properties viewed as essential to "agency".

4.2.2.a A survey of existing agent properties

A list of ten properties defining agency is compiled from five significant publications discussing the nature of agents (see Table 4-1). These properties are now summarily described to allow better comprehension.

- **Delegation**: The concept of agents is intimately tied with the notion of delegation The usefulness of agents comes from their ability to carry out one or several specific tasks on behalf of others. It can be on behalf of a human user but also on behalf of another agent. Regardless of who the beneficiary of the agent activity is, the act of delegation must be an explicit decision **[96]**. Moreover, before making the decision to delegate a task to an agent, one must weigh the risk that the agent will do something wrong against the likelihood that it will do it right **[101]**.

- **Communication**: Communication (or social ability) is a key concept for agency. An agent needs to be able to interact with the user or other agents by using an expressive communication language **[102]**. If nothing else, communication is needed for an agent to receive or send delegation instructions, and possibly to report on the success or failure of actions undertaken while performing a delegated task. Indeed, for all but the simplest tasks, it is necessary for both parties to enter a two-way feedback (discourse) in which mutual intentions and abilities are shared before some sort of contract is passed defining what is to be done and by whom.

- **Autonomy**: An agent must be able to take pre-emptive, spontaneous and independent actions to the extent of the user's specified delegation. It operates without direct intervention from the user or other agents. Once a task has been delegated to it, the agent can pursue its own agenda independently from others and has some control over its actions and internal state.

- **Perception**: Agents live within, and are part of, an environment of some sort. It might be a physical world, a virtual world or a collection of other agents. Whichever, agents have some sensory connection with it. The perception of their environment can be partial and is needed to perform tasks autonomously (in response to specific changes).

- **Action**: Agents must be able to affect their environment via an actuator mechanism. The concept of action, is based on the fact that the agents carry out actions which are going to modify the environment and thus their future decision making **[98]**.

- **Pro-Action**: Agent behaviour tends toward satisfying its objectives (which are the result of a task delegation). Its behaviour depends on its perception, its internal representations (if any) and the communications it receives. Pro-action represents the driving force behind an agent's autonomous actions.

- **Degradation**: In some cases, an agent is unable to completely perform a task it was assigned. This failure can be due to communication mismatch, inadequate resources, or conflicts with others agents. In such cases, agent's behaviour should degrade gracefully and allow most of the task to be accomplished rather than failing to accomplish it all together **[101]**. Such degradation can enhance trust in the agent's performance

- **Learning**: Ideally, there should be an element of learning in the agent behaviour. The agent should be educable in the task at hand and how to do it. The agent can learn from its experience or by "watching" the user. The term "learning" is used to designate high-level functions that affect the way agent operate. It is not related to the learning of the environment through perception (see representation).

- **Representation**: An agent only has a partial internal representation of its environment or none at all. The representation is built over time using both perceptions and communications. It can be used in conjunction with pure perception to decide of the best course of action.

- **Co-operation**: The agent is essentially collaborating with the user (or another agent) in constructing a contract. The user specifies what actions should be performed on his behalf and the agent is providing a list of its capabilities **[101]**.

Table 4-1 summarises the distribution of these properties found in relevant publications discussing the nature of agency (note: the last row, represent the stand this thesis takes on the definition of agency).

	Delegation	Communication	Autonomy	Perception	Action	Pro-action	Learning	Degradation	Representation	Co-operation
In [96]	✓	✓	✓	✓	✓	✓				
In [97]	✓	●	✓	✓	✓	✓				
In [98]	●	✓	✓	✓	✓	✓			✓	
In [99]		✓	✓	✓	✓	✓			●	
In [101]	✓	✓	✓	●	●		✓	✓		✓
	✓= explicitly listed property ● = implied property									
This thesis	✓	✓	✓	✓	✓	✓			✓	

Table 4-1: Essential properties for agency

An analysis of this table allows detecting of consensus areas as well as more individual views on agents. The community is unanimous to recognise *Communication*, *Autonomy*, *Sensing* (perception) and *Acting* as essential to the definition of agency. To a lesser extend, *Delegation* and *Pro-action* also obtain a large consensus on their importance for agents. The other properties on the table appear not to be widely recognised as significant and represent domain specific concepts. It should be noted that other domain specific properties appear in the literature discussing the nature of agents that are not represented in the table. For example:

- **Mobility** is the ability for a software agent to transport itself to different machines through network connections.
- **Veracity** is the assumption that an agent does not knowingly tell lies.
- **Rationality** is the assumption that an agent will try to achieve its goals and will not knowingly take actions that would prevent its goals being achieved.
- **Benevolence** is the assumption that agents will help one-another whenever they are asked.
- **Temporal continuity** is the assumptions that an agent remains active over-time, unlike the one-off execution of most normal programs.
- **Believable** agents are agents that provide illusion of life.

- **Predictive** agents are able to make prediction about the future. In particular they can predict the consequences of their actions.

4.2.2.b Properties justification

The last row of Table 4-1 presents the author's perspective on properties an entity must possess to be considered an agent. This choice, which is strongly validated by the opinion of other experts in the field, is now justified. Yet, it should be noted that the author's choice of defining properties reflects his interest in purely software agents (programs) rather than physical agents (robots). Therefore, the arguments used in this section may not always apply to physical agents.

It is a general trend, not only in software applications, to automate time-consuming repetitive tasks so that humans can concentrate on more interesting complex tasks where their qualities can be put to better use. The extent of this **delegation** of simple repetitive tasks to automated systems depends on the degree of trust the human element can put into these systems and the associated cost if an error is made; effectively risk analysis. For example, in a car, we don't yet trust a computer to drive us around but we are perfectly comfortable with having a fully automated air conditioning system. This comes down to an evaluation of risk and trust in the domain of interest. Before delegating a task to an automated system, one has "*to balance the risk that the agent will do something wrong with the trust that it will do it right*" **[101]**. Failure to regulate the temperature is an acceptable risk and even a simple device will be trusted to do the job. On the other hand, one will need a very high level of trust into an automatically driven car before taking the risk of crashing in it.

Arguably, conventional computer programs have long been performing delegated tasks for users. However their contribution is limited to performing fully specified tasks when requested by the user. From the perspective of delegation, agents are a powerful new paradigm for many software systems especially due to their ability to take **autonomous** decisions. Their ability to schedule their own agenda inside the application they inhabit makes them an ideal candidate to assist the human user in simple and/or repetitive tasks. Applications can be found where autonomous agents can be trusted to automatically perform tasks on behalf of the user. In the current state of agent technology, these domains should possess a moderate level of complexity and the penalty endured in case of failure should remain low. Task delegation does not have to be limited to a user/agent relationship and additional benefits can be reaped from allowing agents to further delegate tasks and sub-tasks to agents possessing more adequate skills.

Indeed, agency really reaches its full potential in "social" situations, where a community of agents lives together inside a system. Inside a community, agents are able to co-ordinate their individual activity into a coherent global system. Note that co-ordination differs from co-operation described by Foner in **[101]** (see section 4.4) which is not thought necessary to agency. This self-organisation is the result of individual behaviours and can rely on two sources of information. Namely, peer-to-peer inter-agent **communication** and environment **perception**. Perception represents the ability of agents to sense their environment, and usually has limited range. Communication abilities make agents, social entities capable of purposefully interacting with others inside their world. In the case of software agents, communication is particularly crucial as it is often used as the "physical" medium of perception.

Agents are different from conventional programs because they are capable of **autonomously** performing tasks delegated to them. It means that agents do not simply execute predefined actions at predefined times. Instead agents **pro-actively** seek the realisation of some internal goals. Such internal motivation makes all the difference between traditional background processing and agency. Indeed agents often (not always) maintain some internal **representation** of their environment, which they use to assess the realisation of their goal, and to make informed decision on **actions** to take in order to reach them.

To summarise, agents are entities that can perform tasks on behalf of others. They live inside an environment that they can perceive and act upon. They possess internal representation of their environment (not always) and internal motivations that are used to take autonomous, pro-active actions. Most importantly, they are social entities, capable of peer-to-peer, asynchronous communication and displaying co-ordinated activity with other agents. This view of agency is shared by many, including Mayfield et al. who argue that the concept of agents refers to their *"ability to: communicate with each other using an expressive communication language; work together co-operatively to accomplish complex goals; act on their own initiative; and use local information and knowledge to manage local resources and handle peers requests"* **[103]**.

4.2.2.c Agency is not Anthropomorphism

There is a common misconception among people discovering agent technology that agency goes hand in hand with anthropomorphism **[104]**. Although an agent-based system can display anthropomorphic properties there is no correlation between the two. More

importantly, it is possible that focusing on the anthropomorphism displayed by some agent-based system tend to hide the full potential of agency.

4.2.3 The Insect Colony Analogy

The "insect colony" analogy **[11, 104, 105, 106]** is commonly used to describe multiagent systems. It can be used both to explain numerous concepts related to agency as well as truly demonstrate the potential of MAS. The insect colony offers a concrete frame of reference for commonly used concepts in MAS. In particular it offers a graphical description of two levels of granularity that a MAS exhibits. This thesis shall be no exception to the rule and use the ant colony analogy.

At the micro-level, it is easy to see each individual ant busily fulfilling its duties inside the colony. A close examination show a fascinating yet simple organism that can sense its immediate surrounding, perform simple tasks and communicate with its peers through the use of scented chemicals called "pheromones". Individual ants can perform very basic tasks such as moving eggs, food or material around; assemble material into simple walls; fight against aggressors if necessary. An ant perceives its environment through touch and smell, which allows a very reduced field of perception in its immediate surrounding. Communication between ants is achieved by mean of secretion of scented chemicals. It should be noted that this mode of communication only allows for an extremely reduced vocabulary.

At the macro-level one can see the complex system that is the colony. It is a self sufficient system able to build its own shelter, collect its food, take care of its eggs, defend itself against aggression and adapt to changing environmental conditions. To say the least, it is a very impressive achievement considering the simplicity of the colony microelement: the ant. The most impressive property of the colony is the self-organisation and order that seem to naturally emerge from the underlying apparent chaos. Indeed when observing the colony at work our judgement on its operations swings between chaotic and organised depending on the level of magnification. The colony is a very robust and flexible system that organised itself from the bottom up rather than relying on a top down chain of command.

The dichotomy between micro-level and macro-level, apparent chaos and order is a striking feature of insects colonies. It demonstrates how simple microscopic behaviours can combine and contribute to the creation of an emergent behaviour at the macroscopic level. This is a clear demonstration of the "all being greater than the sum of the parts". It must also

be emphasised that no global macro-level vision of the colony is necessary to guarantee its smooth operation. Every action at ant level is taken on the basis of local conditions without information on the global situation or goal to achieve.

The insect colony analogy allows for the introduction of required agent properties such as environment sensing, communication and pro-action. More importantly, it demonstrates the principle of coherent emergent behaviour as the result of interactions between simple entities with limited local knowledge of the world.

4.3 Agent Classifications

Presenting classifications of agents according to different criteria is a good way of surveying different area of interest in the domain. Indeed it allows for considering agents from different points of view. Each classification requires that the specific criteria used be defined thoroughly before a taxonomy be presented. This section presents five such classifications based on important aspects of agency.

4.3.1 Taxonomy based on nature

The agent definition given earlier in 4.2.1 is quite generic and applies equally well to physical and virtual agents.

4.3.1.a Physical agents

Physical agents are material entities such as robots. They live inside our physical world (space + time) and, like all other physical objects, they are subject to the laws of physics (gravity, inertia, and chemical reactions). They might be motivated by the realisation of a physical goal and can physically interact with the real world in order to achieve it. Physical agents must be equipped with both sensors and actuators to fit the definition of agency proposed in 4.2.1. Sensors are necessary for the agent to be able to sense its environment and can also be used a physical media for the required communication abilities. Actuators must also be present to allow the agent to modify its environment. A physical agent collects data about its environment through its sensors, applies some forms of cognitive process to generate an adequate course of action and uses its actuator in order to conduct generated actions.

4.3.1.b Virtual Software agents

Virtual agents are not physical entities and exist only within virtual world. Virtual worlds are commonly created inside computers and inhabited by software entities (programs). Therefore notions of virtual agents and software agents are used interchangeably.

Although sharing common concepts, major differences exist between physical and virtual agents. First, in the absence of a physical environment software agents need to be provided with a supporting virtual world. This world can either be specified extrinsically or intrinsically.

- **Extrinsic world** representations reside outside all agents. They are the software ether that exists independently from its inhabiting agents. This approach provides a welcome separation between environment and agents. It also guarantees unity of the supporting world. However, this approach also creates difficulties. A unique world representation can represent an operating bottleneck, as all agents need to sense and act upon their environment simultaneously. Moreover, distributed systems cope badly with extrinsic world. In such cases, maintaining physical unity1 of the world representation degrades performance while distributing it creates synchronisation problems.

- In some cases it is possible to give up such extrinsic environment representation in favour of **intrinsic world** representations. A system fully defined in terms of agents does not require that an external representation exist since all the information is already held inside agents. Agents obtain information about their environment by querying other agents in the systems and can build an internal representation of their local surrounding.

A common problem to both extrinsic and intrinsic representation is the difficulty to describe continuous (and possibly infinite) worlds. Information constituting the virtual world is stored in digital form using discrete physical medium (memory or disk). The amount of storage space available in any computing system is finite, uni-dimensional and discrete in nature. Although programming technique such as arrays and pointers allow organising it into multidimensional spaces, it remains discrete and finite. Discrete worlds can be efficiently represented using arrays of values. Agents can access such extrinsic arrays for *sensing* and *acting* upon their world. No straightforward representations exist for continuous worlds

[1] Physical unity refers to the storage of the representation on a single physical storage medium, not to the nature of the world that remains virtual.

however. Only discrete occurrences can be stored, thereby an additional computing entity must be used to simulate the continuity. Taking CAD applications as an example, a 3D kernel provides the illusion of a continuous three-dimensional world while only storing discrete entities (points, edges, faces, and volumes). The necessity for a computing "wrapper" entity providing the illusion of continuity for non-discrete worlds makes intrinsic representations the logical choice. Since a wrapper must be implemented around discrete storage, agent wrappers are best suited for MAS.

The second major difference between physical and virtual agent is in the role of communication. While it has been proven that physical agents can co-operate without communication **[107]**, this is only possible because of the their ability to use physical sensors to collect environmental data. Although one can imagine implementations of MAS using agent accessing extrinsic world representations directly (through memory read/write), it is not a realistic approach. Most virtual agent must rely on communication no only for inter-agent relationship but also for both *sensing* and *acting* upon their environment. Therefore communication capabilities are paramount for software agency (see section 4.2).

The main interest of this thesis is in software agents rather than physical agents, therefore for this point onward, unless otherwise specified the term agent will be used interchangeably with software agent. Equally, multiagent systems (MAS) will designate software MAS unless specified as physical.

4.3.2 Taxonomy based on sociability

It has been seen, that agency goes hand in hand with society. Indeed, agents are communicative creatures that base a large part of their activity around information collection and exchange inside a community. However, one can distinguish two types of agents based on their sociability.

4.3.2.a Standalone Agent

Standalone agents perform a useful and complete task on their own. It does not mean that standalone agents do not communicate with other agents. These are independent entities capable of performing entire tasks by using only their internal know-how. Communication is still used to collect data and generate actions however.

Example of typical standalone agents are programs known as "desktop agents" or "Intranet/Internet agents" **[96]**. These agents interact at user level and perform typically

repetitive tasks on behalf of the user that can be conducted unattended. Various computing tasks answer this description:

- Interface agents can monitor the user's input, learn from it, provide assistance for complex tasks (wizards), suggest alternative work methods or automate repetitive tasks (input patterns).
- Mail filtering agents automatically handle incoming e-mails in ways specified by or learned from the user.
- Data gathering agents can roam a network autonomously and notify the user for potentially interesting data and updated sources of information. They can also perform extensive searches for specified criteria.
- Operating system agents can perform low-level maintenance tasks on operating systems such as disk de-fragmentation and virus checking.
- Application specific agents provide various high-level services to users within the application itself. Most types of applications can benefit from agent technology. For example, databases, office applications and CAD systems are among potential agent-enabled applications.

It should be noted that all the examples cited above might also be automated using more conventional techniques. Moreover, it is a recognisable trend in the IT industry to use the term agent for software that would not be recognised as such in the scientific community **[6]**. In particular operating system "*agents*" have been commercialised that merely use time-based scheduling of administration tasks. Identically, various available Internet "agents" simply do not fit our definition of agency. However, this should not undermine the potential value of applying agent technology to these areas of computing activity.

4.3.2.b Multiagent Systems

Multiagent systems perform tasks by combining the knowledge and know-how of multiple agents inside a community. Unlike standalone agents, individual agents inside MAS usually have no meaningful capabilities at user level or do not possess the entire know-how necessary to conduct their task. It is the inter-relationships and co-ordination between a community of such agents that creates a global useful system capable of performing tasks on behalf of the user.

Multiagent systems: ① **Localised MAS**

One possible physical[2] configuration for MAS is that of a localised community of agents. All agents exist on the same computer, possibly inside a unique process or application. This configuration is commonly used for multiagent simulations and in problem solving using multiagent techniques. It is particularly well suited for problems where strong inter-dependence exists between agents in the system. Such problems tend to generate high communication load inside the agent community used to represent them. In today's computers, memory accesses are measured in nanoseconds, network connections in milliseconds. This great difference of speed is incurred due to the necessary computational overhead introduced by network communication protocols. Therefore, by keeping the entire agent community on a single machine, it is possible to greatly reduce the performance cost of communication by avoiding slow network connections and using direct memory access as the communication medium between agents.

Multiagent systems: ② **Distributed MAS**

An important strength of multiagent approaches is their potential for parallelisation and distribution across multiple computers through networks. Agents represent independent, autonomous computing entities that can be used as a basis for distribution across networks. By running agents on separate machines, one creates truly concurrent processing and increases the computing power available to the multiagent system. However, it has been discussed earlier that network connections necessary to support inter-agent communication can represent a major performance bottleneck. Therefore, distribution of MAS is not adapted for system with high inter-dependence between agents. Instead, it can be efficiently used in agent communities that are loosely inter-connected. In such communities, the benefit of additional computing resources and true concurrency far outweighs the penalty paid for communicating between agents.

Multiagent systems: ③ **Agent granularity**

The decision on whether to distribute a MAS or not also depends on the granularity of the agents concerned. Because a strong correlation exists between the granularity of agents and their inter-dependence inside a model, it can be used to guide the distribution decision.

Generally speaking, it is possible to classify agents using their granularity. Fine-grained agents represent basic computing entities performing simple tasks on numerous pieces of

[2] In the case, "physical" refers to the hardware supporting software agents.

data. They generate high communication load within their community, which should therefore be kept localised. Coarse-grained agents provide high-level services, perform complex tasks using relatively few abstract pieces of information. They are good candidates for distribution since their dependence to the rest of the community is less than fine-grained agents. It should be noted that real MAS do not usually fall completely in one or the other category. Instead, they contain agents of different granularity organised in hierarchical structures. This allows for adaptive distribution of MAS based on agent granularity.

For example, let's consider an automated assembly line composed of three robot arms needing to be automatically controlled. High-level agents providing services such as grabbing parts from a conveyor belt, bringing parts into an assembled position or avoiding collision with other robots can represent each robot arm. Each robot can also be represented as MAS where low-level agent represents its constituting sub-components (hand, forearm, and arm). These two agent-based approaches to robotics are further discussed in 4.3.4.d. However, a hierarchy of nested MAS can be used to model the system. Distribution of these MAS is decided according to the granularity of agents. Low-level agent must be kept local, while high-level agents can benefit from being distributed across several computers as illustrated in Figure 4-1.

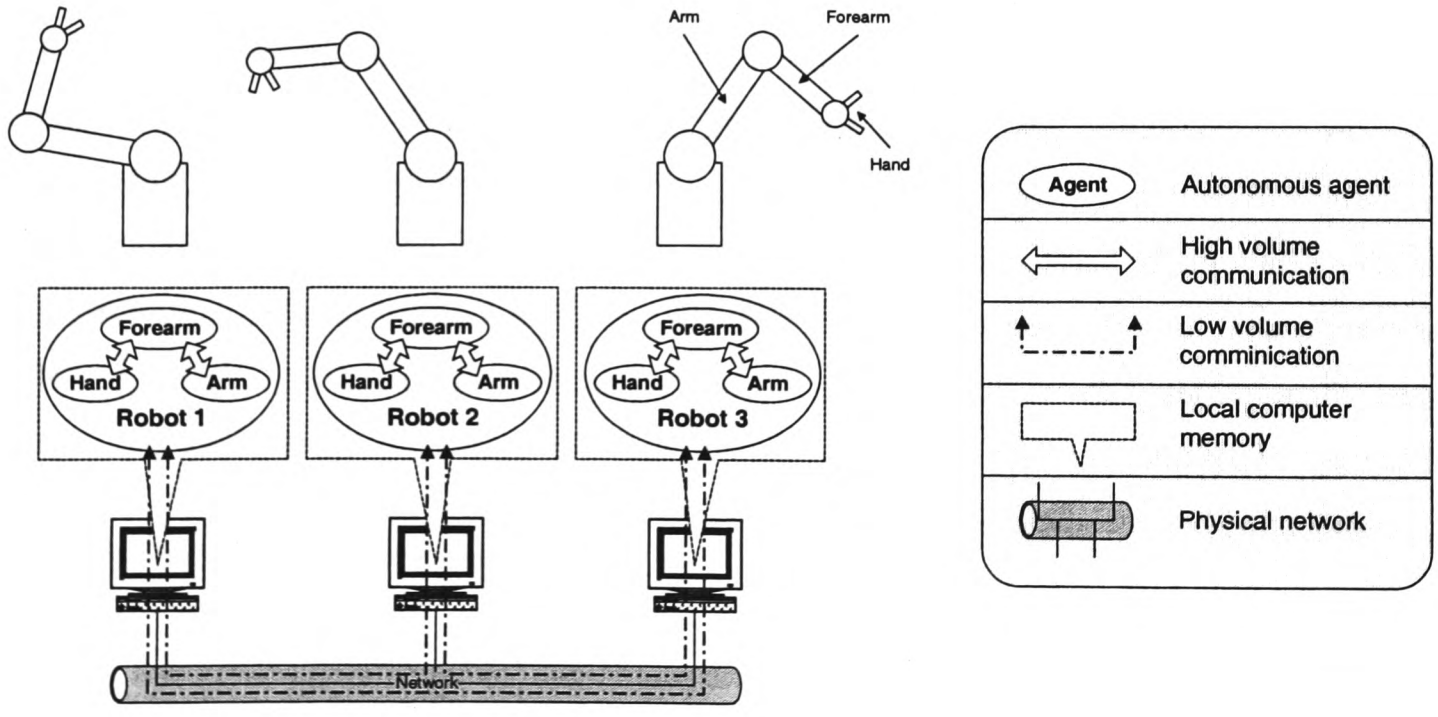


Figure 4-1: Mixed nested MAS – Localised and Distributed

Figure 4-1 shows that MAS can be nested into one another other and also how localised and distributed can co-exist within the same system. Each robot agent is itself a MAS constituted by basic elements. The basic agents handle low-level movement co-ordination

within a robot, while high-level robot-agents deal with advanced tasks possibly involving another robot (passing of objects).

4.3.3 Taxonomy based on rationality

Agent rationality describes the reasoning mechanism used by agents to generate their actions and behaviours. Two categories are identified in the abundant literature concerning agents; *deliberative* and, *non-deliberative* **[99, 100, 108, 109]**. Some hybrid approaches attempt to combine the benefit of both.

Deliberative agents have complete and up-to-date knowledge about their environment. They take decision by explicitly deliberating upon various options and evaluating their potential for reaching the agent's goal. Deliberative agency has its root in the *sense-plan-act* AI school of thought "*which aims at producing provably correct sequence of actions, or plans, which upon separate execution would have the effect of achieving desired goal state*" **[108]**. Although efficient in static situations, forward planning is less adapted to dynamic situations (You cannot plan for the unexpected!). To accommodate planning in rapidly changing environments, some deliberative agents can generate alternate plans and execution strategies. Monitoring of execution allows for plan modification or complete re-planning. Despite these efforts, deliberative generation of behaviours is often computationally expensive (therefore time-consuming) and its reliance on up-to-date world representation adds to the complexity of deliberative approaches.

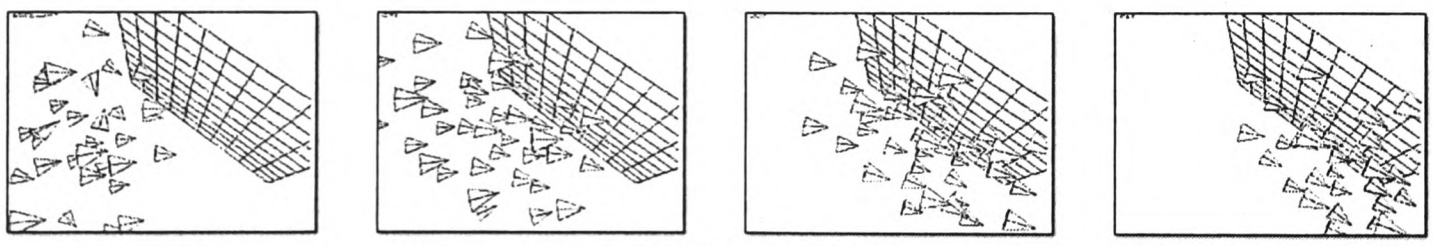
Problems encountered in deliberative agency have led to the development a new agency paradigm that adopts a radically different approach to decision making. In non-deliberative agency (often referred to as *reactive* or *situated*), decisions are typically taken at run-time, without forward planning and using only limited knowledge (if any) about the environment. Such agents are not required to elaborate correct plans based on complex reasoning. Instead, they must produce robust actions that are appropriate in encountered situations. In **[109]**, Werner strongly opposes the concept of *non-deliberative* agency on the ground that complexity and organisation cannot spawn from simplicity and chaos. Yet, multiple applications have shown *non-deliberative* agents to be efficient in a variety of situations, proving that further theoretical research is required in this area. Regardless, *non-deliberative* agents are simpler and faster but do not offer the same flexibility or adaptability as their *deliberative* cousins.

Decision making in agents is a core problem in autonomous agency. Two contrasting approaches exist. On one hand, *deliberation* attempts to make as many decisions as possible as far ahead in the future as possible, which requires complete knowledge of the situation and complex reasoning capacities. On the other hand, *reaction* delays the decision making to the last possible moment when most of the information needed is present in the environment and simple solutions exist. Neither can pretend to answer all questions, and hybrid systems are often required. A classification of autonomous agents is now proposed based on the type and complexity of their decision mechanisms.

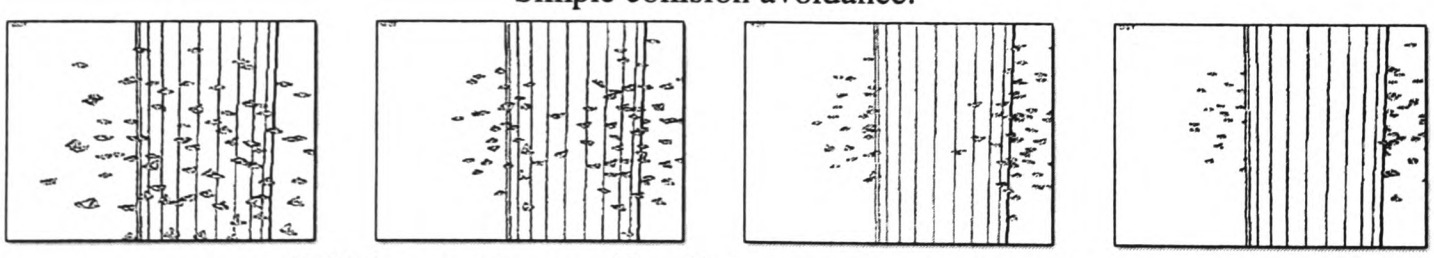
4.3.3.a Reactive Agents

The simpler method for generating behavioural responses is the stimuli-reaction method. Agents are equipped with a set of pre-defined responses to external stimulation. In practice, reactive agents monitor their environment for particular situations or events and apply pre-defined static rules to generate appropriate responses. Because reactive agents do not plan their actions in advance, they do not require an internal world representation to function. Indeed, they can completely rely on the environment to provide stimuli to trigger their internal reaction rules.

① *Reactive agents:* **Grou p behaviours**



Simple collision avoidance.



Collision avoidance with splitting and merging behaviours.

Figure 4-2: Flock behaviours in Boids (from [111, 112])

Although very simple, the reactive approach is powerful and allows for modelling of complex problems. Consider the motion in space of a swarm of insects, a flock of birds or a shoal of fishes famously studied by Reynolds in **[111, 112, 113]** and illustrated in Figure 4-2. These are spatial aggregation of autonomous entities moving together in their environment. These formations display very interesting collective behaviours, for they seem to move in

harmony, as if controlled by a common instrument. They can perform various tasks such as avoiding obstacles or evade predators while always remaining globally co-ordinated. However, the cohesion of such groups is not guaranteed by a centralised organisation, but through the individual actions of each individual. Conventional top-down modelling techniques might be applicable for small groups but would struggle to keep up as their size increases. The bottom-up, agent-based modelling is perfectly adapted for such cases. These are typical examples of complex systems that can be efficiently modelled as reactive MAS.

The motion through space of these groups is not a planned activity. Instead, each individual within them reacts instinctively and instantly to their local surroundings. No internal mental state is required and agents can rely entirely on their ability to monitor their immediate surroundings. Individuals throughout the group apply simple stimuli-reaction rules, which lead to a coherent global behaviour. For example, Mesle showed that aggregating behaviours in fishes could be obtained using a single rule applied by all individuals **[114]**. This single expression base on vector fields takes into account the current speed, direction and position of the individual and its close neighbours to yield a new direction and speed to be followed. Other methods involve the use of attraction and repulsion force fields to create motion tendencies in agents **[115, 116]**.

Other group behaviours have been successfully modelled using reactive agents. In **[104]**, Parunak give examples of complex group behaviours obtained using agents and simple stimuli-reaction rules. Lee also surveys potential applications of reactive agency before presenting various reactive architectures for agent systems **[117]**.

② *Reactive agents:* Mark ing the environment

A noticeable technique used in reactive agency is the marking of the environment as a mean of co-ordination. It has just been shown that vector and force fields can be used efficiently to co-ordinate motion in space. However, such co-ordination lacks any important purpose other than the motion itself. By allowing agents to leave marks on their environment, researchers can generate global co-ordination with a purpose. In **[104]**, Parunak describes particularly well how pheromones left on the environment by insects can lead to global co-ordination of the colony in food collection. Consider these simple reaction rules applied by individual ants in search for food:

```
IF (carry nothing) THEN (roam AND look for food)
IF (find food) THEN (pickup food AND drop pheromone)
IF (carry food) THEN (go to nest AND drop pheromone)
IF (in nest AND carry food) THEN (drop the food)
IF (hungry AND tired) THEN (go to nest)
```

While carrying nothing, ants roam the land away from the nest in search for a source of food. In the absence of pheromone in their immediate surrounding, ants roam in random directions. If pheromone is detected however, roaming is less random and ants tend to go in its direction. Because only ants carrying food mark the environment with pheromone, they mark paths from food source to the nest that can be followed by others. Pheromone is a degradable product, thereby, unused paths disappear and busy ones are strengthened. A self-preservation rule prevents ants starving during unsuccessful roaming by making them return to the nest eventually. Five simple rules and the use of pheromone markers allow ant colonies to efficiently harvest existing food sources until exhaustion, and prospect the environment for new ones. The marks generated by ants carrying food accomplish two functions;

- **Co-ordinate actions:** by showing the way to food source, markers reduces random roaming and creates globally organised activity.
- **Optimise performance:** positive reinforcement of profitable path helps concentrate the effort of the colony on the best sites and reduces fruitless roaming.

Similar marking techniques are commonly used in collective robotics **[118]**.

4.3.3.b Motivated Agents

Reactive agency was shown to be an attractive solution because of its simplicity. Actions are taken "instinctively" in response to external stimuli without knowledge of past actions or potential consequences. These limitation means it can only handle situation where forward planning is not necessary. It is thereby not applicable to a wide range of problems where multiple actions must be performed in order to achieve a goal and problems in which previous events should influence present behaviours.

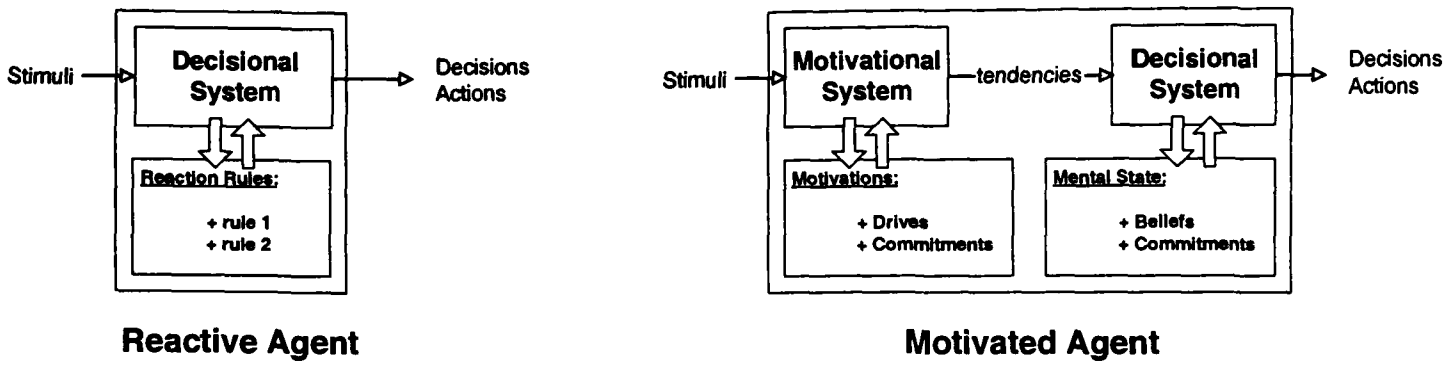


Figure 4-3: Decision making in Reactive and Motivated agency

Motivated agency refers to agency in which behaviours are not automatic responses to environmental pressures but also the result of internal drives (or motivations). In **[98]**, Ferber defines motivation as "*the reasons which push an agent into action, [...], which serve as the basic material for the constitution of tendencies*". The generation of tendencies rather than straight-out actions is a critical difference between reactive and motivated agency. Figure 4-3 shows how motivated agents introduce an additional level of processing (the motivational system) compared to reactive agents. Reactive agents receive information from the environment, check their reaction rules and generate actions if a suitable rule exist. Motivated agents use information from the environment and natural drives to generate tendencies that are passed to a decisional system. The decisional system filters the generated tendencies and yields final decisions.

Motivated agents rely on internal representations for the decision making process. These representations constitute the agent's mental state and contain two different types of information.

Motivated agents: ① Mot i vations

Ferber distinguishes four types of motivations; namely personal motivation, environmental motivation, social motivation, and relational motivation. They all contribute to the global motivational system that creates internal tendencies. Some motivations influence agents positively by encouraging agents to take actions, others negatively by inhibit agents' behaviour.

- Personal motivations, or drives, tend toward the satisfaction of the agent's needs and the realisation of commitments it has with itself. Agents can not remove themselves from their drives for they are internal stimulus produced by them. Typical examples in biological entities would be thirst, hunger and sex drive.

- Environmental motivations are the result of what agents perceive of their environment. They are the source of all reflex actions but also give assistance to other motivations. For example, the perception of food might trigger dormant appetite.

- Social motivations relate to pressures exerted by the society agents live in and also by the designer of the MAS. In particular it expresses the function and duty of agents in society.

- Relational motivations represent commitments that agents can have with others inside the community. They are essential to cognitive agents capable of anticipating the future for it help reducing the degree of unpredictability in MAS. Indeed, by committing to a task, agents agree to restrict their potential future behaviours, therefore allowing others to plan their actions accordingly.

Motivated agents: ② Beli efs and world representation

Reflex actions taken by reactive agents are triggered uniquely by perceived information from the environment. Reaction rules representing internal motivations are applied in response to specific stimulus but agents lack any knowledge of their past, current or future situation. The realisation of motivated action is not possible without the existence of an internal representation of the agent's environment. Indeed, goals are typically expressed in term of a desired state of the agent's environment.

Take the example of a football player motivated to winning a match. This simple motivation can be expressed by a simple rule:

$$N_s \rangle N_c$$

Where N_s is the number of goal scored by his team and N_c the number of goals conceded by his team. The player's behaviour is to go on the offensive if losing or drawing and to play defensively if already in the lead[3]. Monitoring scoring events is not enough to assess the current situation. This simple goal requires that the player remember the number of goals scored and conceded in order for his motivation to be activated.

Therefore, motivated agents must store internal symbolic representations (beliefs) of their environment. It is used during operation for assessing the level of achievement of the agent's

goal. This symbolic content is created through the perception capabilities and must be kept up-to-date for it to remain useful to the agent's activity.

Several types of beliefs can be distinguished: *collected* facts represent information entering the agent through his sensory system and should be accurate reflections of the environment. *Communicated* beliefs are beliefs entering the agent through communication with other agents in the system. Communicated beliefs are usually trusted to be correct although the possibility of deceiving agents is a possibility that sometimes need to be addressed **[101, 102]**. Beliefs can be requested from and by other agents through communication or volunteered to the community through broadcast mechanisms. Finally, belief can be *deduced* through internal reasoning. The accuracy of this last category is dependent on the quality of the agent's reasoning and its applicability in given situation. Accurate beliefs are critical in the decision process of agents. Thereby, the maintenance of reliable, truthful beliefs is the subject of active research **[119]**.

4.3.3.c Planning agents

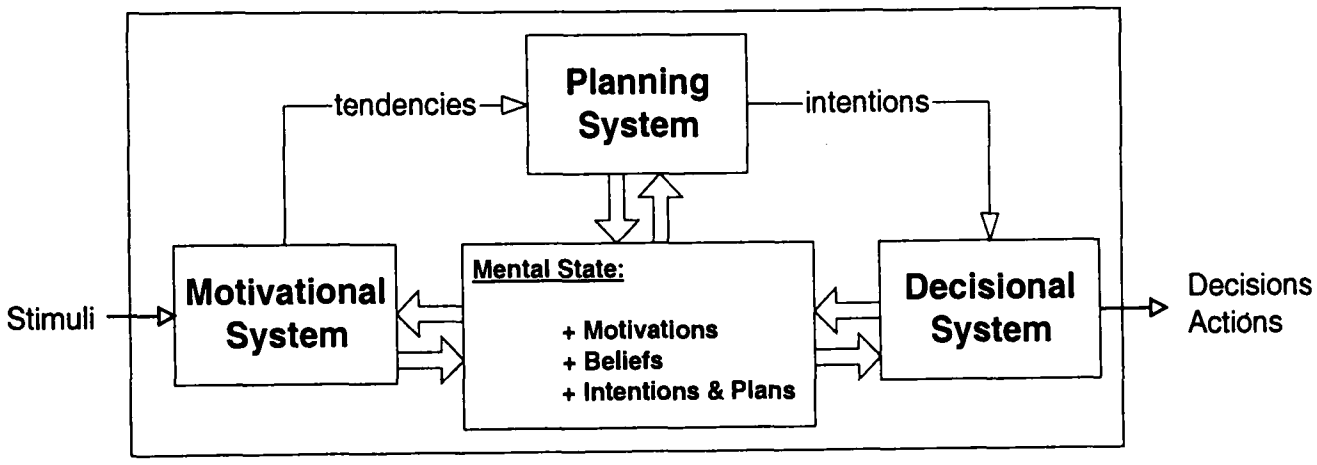


Figure 4-4: Decision making in planning agency

Planning (or deliberative) agents add an additional level of processing to motivated agency in the generation of behavioural actions (see Figure 4-4). They introduce the notion of intention, which represent a decision taken by agents to perform an action in the future rather than immediately. Intentions are combined into plans than agent can elaborate then carry out in order to achieve their goals. Such forward thinking requires agents to assess the potential consequences of planned actions in order to predict future states of their environment. Future prediction can be notably unreliable when made in very dynamic environment.

An important characteristic of intentions is that they can be cancelled before being executed therefore allowing greater flexibility. Indeed, reactive or motivated agents commit

[3] This strategy would be strongly contested by any football fan and only serves as a basic example.

strongly to their decisions **[120]**. Once a course of action has been decided it is carried out to completion regardless of potentially changing conditions. Planning agents have the potential to modify or scrap elaborated plans before or during execution if they become inapplicable.

Planning agents are well suited to situation were actions are not instantaneous and the environment does not change significantly between choosing a course of action and fully executing it. This later condition can be addressed through dynamic re-planning at the expense of higher workload.

4.3.4 Taxonomy based on application

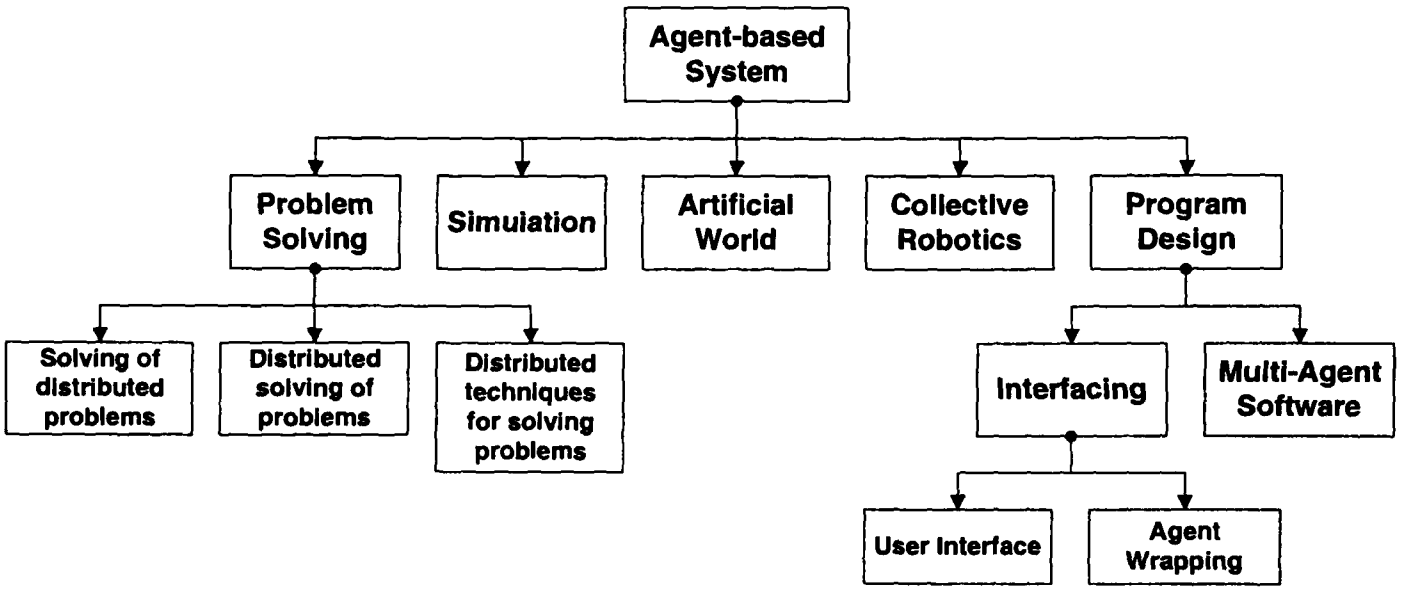


Figure 4-5: Classification of agent-based systems based on application (based on [98])

Areas of applications for agents and multiagent systems are numerous and illustrated in Figure 4-5. Further subsections discuss some of the most significant applications of agent technology.

4.3.4.a Simulation

Simulation is a very active branch of computer science research, which intends in to analyse the properties of theoretical models of the surrounding world **[98, 121]**. Researchers construct theoretical models of reality in order to explain or predict natural phenomenon. Simulation "runs" on computers are commonly used to test the correlation and accuracy of such model with reality. Models are usually set up as a set of complex mathematical relationships (differential equations, transition matrices) between variables representing physical values. Although successfully applied to numerous problems in physics, chemistry, biology, ecology, economy, and others, this approach suffers from limitations;

- Different levels of analysis in a model are difficult to express using mathematical expressions.
- The actions of individuals are difficult to take into account.
- Mathematical simulations handle quantitative data well but are inadequate when dealing with qualitative information.

"*Multiagent systems can bring a radically new solution to the very concept of modelling and simulation in environmental science, by offering the possibility of directly representing individuals, their behaviours and their interactions*" **[98]**.

Multiagent systems allow the creation of *individual-centred simulation* by their capacity to model individual properties and behaviours. The strengths of agent-based simulation are flexibility and integration. Agents can handle quantitative and qualitative information. Their behaviours can be expressed analytically or using rule-based heuristics. Modification of agent-based simulation is done incrementally by adding individual behaviours and properties. Moreover, multiagent simulations allow for the simulation of complex systems where global behaviour emerges from underlying interactions between individual rather than obeying system-wide rules.

4.3.4.b Interfacing

Because of their embedded communication skills, agents have great potential when it comes to interfacing different systems. Indeed, agents are used as interface modules between human being and computers **[122, 123]** and also between computer systems **[124, 125, 126]**.

Interfacing: ① **User inter face**

Agents can be of benefit when interfacing human beings and computers. In particular, the autonomy that agents display can be used to create new types of interaction between user and program. An interface agent embedded inside a program can;

- volunteer information to the user,
- ask the user advice about ongoing activity,
- give advice to the user concerning current activity,
- learn the user's repetitive task patterns by monitoring his activity,
- learn changes in user work pattern,

- generalise user's intent from his actions,
- autonomously execute tasks on behalf of the user.

This sort of autonomy radically changes the relationship between user and program. Traditional interfaces, graphical or not, require the user to fully specify the task that is to be performed. The granularity of that specification varies among applications, but is usually fairly low. The user has to define all aspects of a task (input and output files, type of operation, operations parameters ... etc.) before the application can carry it out. During the specification phase, which represent 95-99% of the time for most applications, the workstation remains essentially idle. Special applications such as FE analysis, CFD analysis and image rendering (especially when using ray-tracing) involve very long period of heavy processing therefore bringing the average idle times to much lower values. However, these computations are typically conducted "over-night" and without direct user intervention. Even in such applications, the idle time is high during the specification stages. Autonomous software agents are a way to harness this wasted processing power for the benefit of the user[4].

"*Agents can transform passive personal workstations into processing entities that actively co-operate with the user, taking advantage more fully of the computational power*" **[123]**.

Indeed, interface agents can use this vast source of available power to carry out tasks that benefits the user. Such agents are typically used in a GUI (graphical User Interface) environment. They monitor the user's activity by eavesdropping events generated within the GUI. The information collected is used in two separate ways. Firstly, it is used to identify new patterns of events to build an internal "mental model" that have some correspondence with that of the user **[122]**. Secondly, they are used to detect previously identified patterns and infer adequate course of actions based on previous experiences. Techniques borrowed from the field of artificial intelligence are commonly used to carry out these two tasks. In particular, in order to remain useful to the user, interface agents must be adaptive and possess learning capabilities that allow them to learn new patterns but also to modify existing one.

[4] Other schemes exist that reap the benefit of idle time using conventional background tasks in the context of distributed computing. SETI@home and distributed.net are examples of such schemes.

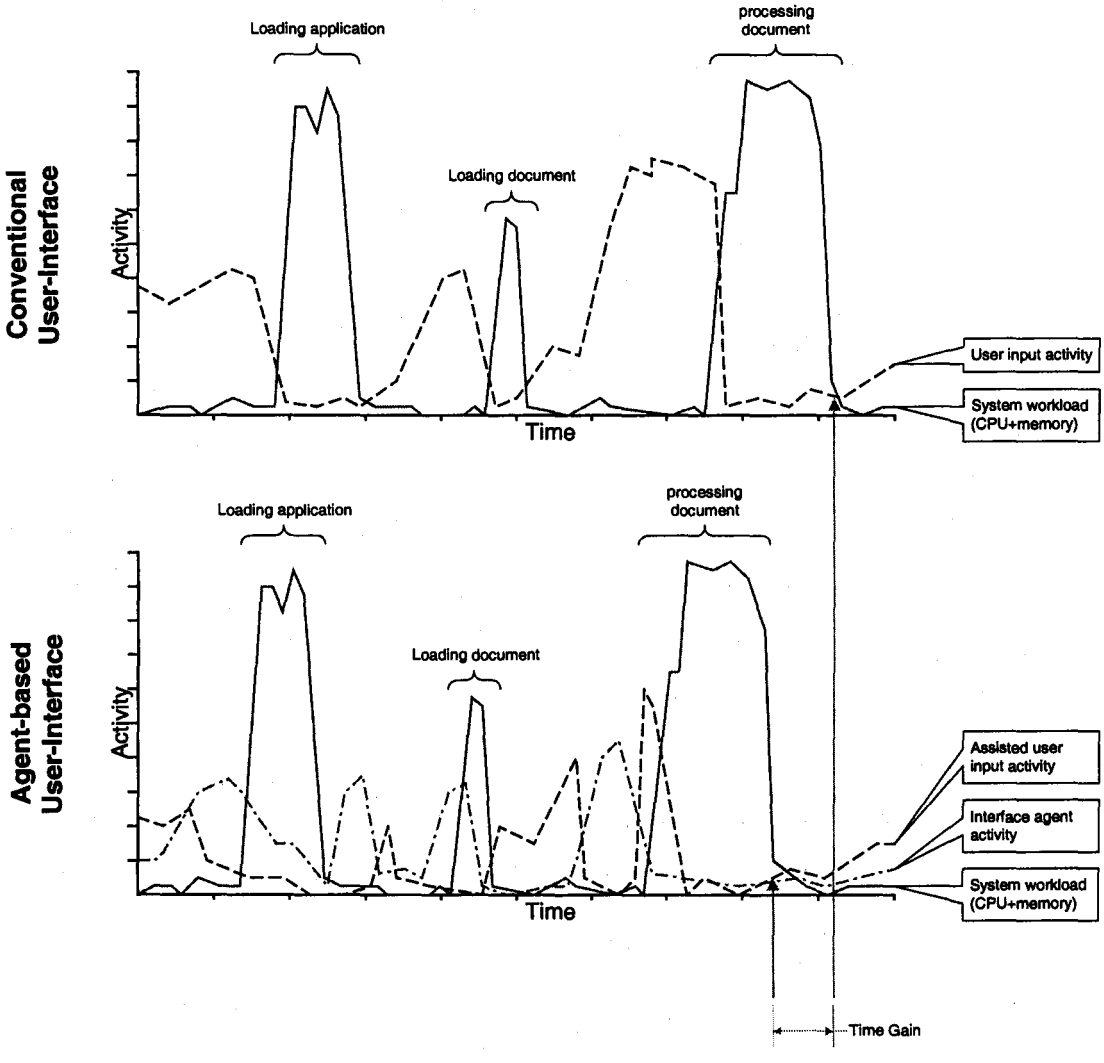


Figure 4-6: Activity in systems with conventional and agent-based Interaces

Figure 4-6 illustrates the activity taking place during a work session on systems equipped with a conventional GUI and agent-enabled GUI. Using the conventional interface, two different modes are apparent. Specification stages involve high workload for the user while the application is essentially idle. Once an operation has been specified (correct options selected, parameters entered) the user launches the command. The system enters a processing phase where system usage is high but user input is minimal. Once the specified operation has been executed and results been obtained, a new specification stage can start. Agent-based user interfaces display less dichotomy. Agent-enabled interfaces use the available processing power to reduce the user's workload. The interface agents can detect known action patterns through monitoring of the GUI. The agent can automatically complete established patterns on behalf of the user, thereby reducing workload and speeding up specification activity.

Interfacing: ② **Agent wra p pers**

Application wrapping (or wrappering) is commonly realised on legacy systems that are required to interface with new systems using different communication and operation protocols. When using conventional programming paradigms, "w*rappering refers to the development of an object-oriented (OO) interface for an application, where the application itself does not offer an OO interface or has an OO interface which is based on a different object model than the developer is using*" **[96]**. For example, such OO wrapping has been extensively conducted on legacy software in the banking sector in order to accommodate new technologies and enable global integration of existing subsystems.

However replace "object-oriented" with "agent-oriented" and the same definition defines agent wrapping. OO wrapping permits the interoperation of legacy system inside new technological architecture such as CORBA (common object request broker architecture) or COM (component object model). Agent wrapping intends to enable the integration of conventional systems inside multiagent architectures. In the most basic form, agent wrappers provide the required high-level inter-agent communication capabilities, thereby enabling other agents to interact in an homogeneous manner with all participating systems. However, the full potential of agent wrapping requires the addition of mental attributes (beliefs, goals, and reasoning capacity) requisite of agency. Fully-fledged agent wrappers allow conventional applications to act as autonomous, goal-driven agents inside MAS. They can initiate communication and actions with other agents, in order to achieve their goals. Frost, in **[124]**, gives a good example of agent wrapping. He describes agent wrapping applied to conventional CAD, CAPP and CAM system. Agent enabled application communicate through CORBA and autonomously exchange data in order to guarantee manufacturability of designs.

4.3.4.c Problem solving

The concise Oxford dictionary defines solving as "*finding an answer to, or an action or course that removes or effectively deals with a problem or difficulty*". In this broad definition, problem solving describes most of agent's application discussed in this chapter. This section narrows this definition to the study of software agents applied to the resolution of computing tasks that are hard to understand, or accomplish, or deal with. This is known as "distributed problem solving" and actually covers two distinct activities; namely the solving of distributed problems and the solving of problems using distributed techniques.

Problem solving: ① **Distr i buted problem solving**

Large complex problems often require an equally large and complex range of expertise to solve. When the scope of a given problem is too wide, it is a natural to divide it into simpler sub-problems to be solved by different contributors. Each contributor can apply its specialised expertise to the task it is allocated. The solved sub-problems are eventually recombined to yield a global solution. However, sub-problems are rarely completely independent and communication and co-operation between them is necessary to reach compatible sub-solution that can be recombined into a global result. Agent technology is well suited to distributed-solving of problems. Indeed, agents represent computing entities containing specific knowledge and skills. They also possess strong communication abilities and are naturally co-operative Therefore agents can be applied to different sub-tasks of a project and bring task-specific solving skills as well as their ability to share results and co-operate towards a common goal.

Consider for example the design of a modern aeroplane. It is a hugely complex engineering task involving numerous highly specialised disciplines such as aerodynamics, structural design, thermodynamics, electronics and so on. Such a huge project has to be organised into more manageable co-operative sub-tasks in order to be realised. This way, experts in various fields are allocated suitable tasks and can solve problems relating to their area of expertise. At a global level, a great deal of effort typically goes into managing the communication between sub-tasks, which are critical to the success of large projects.

A distributed problem of particular interest to the author is the design for manufacture of mechanical components. This is typically a distributed solving process involving CAD and CAPP. Frost has shown how agents can help bridge the existing gaps between the different sub-tasks involve in the DFM process **[124]**. He uses agent wrappers around conventional CAD and CAPP packages to create autonomous data exchanges between them and thereby achieving more efficient co-operation.

Some problems also naturally decompose into physically distributed sub-problems. In particular, control and monitoring of physically distributed system falls in this category. Such distributed problems still display functional distribution but also present a physical distribution in space. For example Chaxel, is concerned with the weaknesses of traditional control methods when faced with the physical distribution of machine tools the shop-floor **[127]**. He demonstrates that agent-based approach applied to control can bring increased robustness and flexibility through the use of embedded memory and communication.

Problem solving: ② **Distr i buted solving techniques**

Agent technology can also be beneficially applied to problems that are not naturally distributed. This is true for problem solving in the classic sense of the term, which is to find a solution to a fully formulated problem. Indeed, agent-based approaches offer novel ways to reason about classic problem. They offer radically new ways of breaking down the solving process by allowing individual protagonists of a problem to "sort themselves out". Agent technology offers a novel and appealing solving tool. Ferber find that their "*advantage lies in their speed of execution. Based on considerations which are very different from those adopted in classic approaches through state space exploration, they make greater use of certain structural characteristics of problems*" **[98]**.

4.3.4.d Collective robotics

"Collective robotics relates to the creation not of a single robot, but of an assembly of robots, which co-operate to accomplish a mission" **[98]**. Ferber distinguishes two agent-based approaches to robotics; namely cellular robotics and mobile robots.

Cellular robotics views robots as built from elementary parts, each implemented as an individual agent **[128]**. For instance a classic serial robot arm could be built from a hand-agent, a forearm-agent and an arm agent (see Figure 4-1). Co-ordination between the constituting robot elements is carried out through agent activity. This approach is particularly interesting as the number of moving elements in a robot increases because the level of complexity remains constant. Indeed, the inter-agent co-ordination protocol remains identical regardless of their number. Motion of a robot arm can be obtained by specifying a desired position of one of the element (the hand for example). The newly motivated agent might be able to achieve it on its own. Otherwise, it will initiate interactions with its immediate neighbours. The motion will propagate along the required elements in the arm to achieve the specified position.

Mobile robotics is concerned with co-ordinating the actions of two robots or more inside a given environment **[118]**. Research in collective robotics is mainly concerned with specifying individual agent behaviours that result in a group of robot achieving one or more tasks. Agents might try to go from A to B, clean the floor, collect items, or assemble a motorcar. Their behaviours vary from simply avoiding each other to entering complex co-operation contracts or even competing with each other. Once again the emergence of global behaviours as a result of interactions between individual agents is a strength of the

multiagent architecture. Designers do not need to specify all possible combinations at a global level and can instead concentrate on local interactions. High flexibility can also be achieved since new agents can be added inside an environment without modifications to the rest of the system.

4.3.5 Taxonomy based on internal architecture

A final classification of agents is proposed based on the internal organisation of agents, commonly called architecture. This taxonomy is not particularly useful as such but it allows the presentation of underlying technologies used in agent-based systems.

4.3.5.a Modular architecture

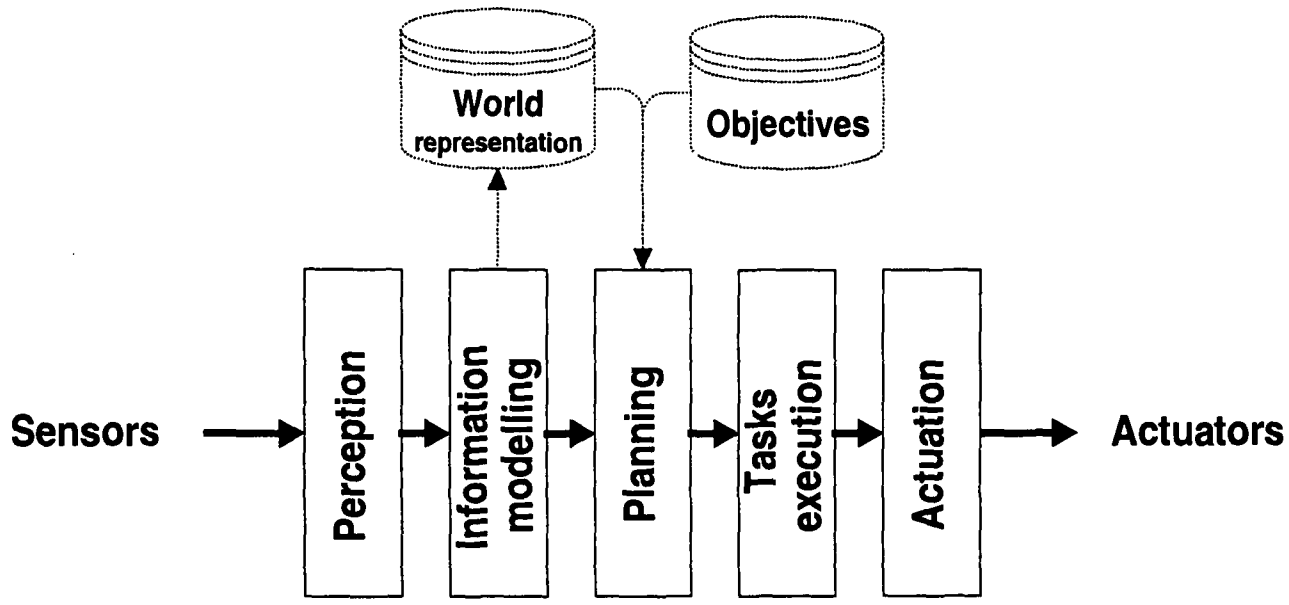


Figure 4-8: Modular architecture

The conventional approach to implementing autonomous agents involves the creation of a series of functional modules handling different aspects of agent's activity. Figure 4-8 illustrates how functional modules are stringed together inside an agent to achieve the desire operation.

Information flows through the different modules in a linear manner. Every module manipulates and transforms the data before passing it over to the next in the chain. The level of abstraction of the data manipulated varies along the chain of modules. During an "ascending" phase from sensing to planning, the data is incrementally abstracted. The refined and abstracted information is used in the decision process (planning). The abstraction level is decreases during the "descending" phase that transforms high-level plans into concrete actions performed through actuators.

This conventional decomposition scheme is mainly motivated by implementation contingencies. The serial flow of information is easier to manage than a parallel one. Planning of all actions is done inside the same modules, thereby allowing consideration of all known parameters before taking any decisions. The main drawback of the modular architecture is its lack flexibility (the addition of new behaviours can require modifications in all modules).

4.3.5.b Subsumption architecture

The subsumption architecture divides agents, vertically instead of horizontally. Each module is responsible for one activity or behaviour of the agent. These behaviour modules are organised in a layered fashion. Low-level modules represent reflex actions, which involve little reasoning and are given priority over higher level behavioural modules as they provide rapid response to time critical situations. Higher level modules represent less critical, more elaborate behaviours.

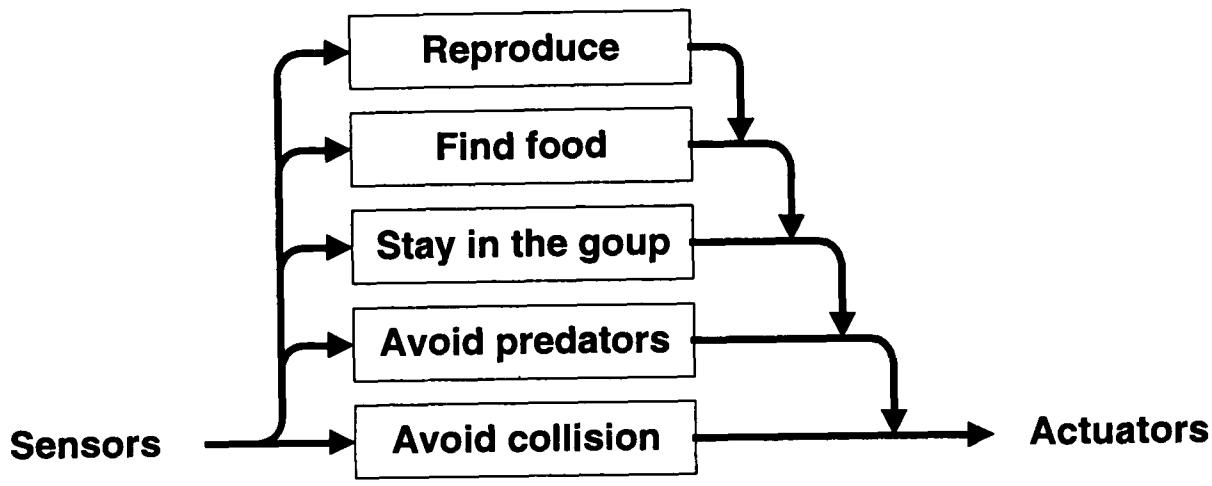


Figure 4-9: Example of subsumption (or layered) architecture

Figure 4-9 shows an example of layered behaviour for a fish swimming in shoal as discussed in 4.3.3.a. The low-level behaviours "avoid collision" and "avoid predator" have priority over all other modules as they represent activities on which depends fishes survival. Higher level activities are added layer by layer in order of priority.

The interactions between modules, defined by priority order, are fixed during design stage. The different modules work in parallel and the system relies on the priority mechanism to choose between any conflicting decisions taken by different modules. This approach pioneered in mobile robotics **[129]** represent a more flexible decomposition of activities inside agents. It provides a way to incrementally build and test agents by adding new

behavioural layers to existing ones. The agent does not require complex central control and can be seen as a collection of individual behaviours each assuming various tasks.

The approach can be used to accommodate various level of rationality inside a unique agent, with lower-level reactive behaviour and higher-level more cognitive (motivated, deliberative) one **[130]**. It is also highly seductive as it provides automatic conflict resolution between behaviours through the priority mechanism. However, a priority order must be chosen even in situation where behaviours are equally important.

4.3.5.c Blackboard architecture

The blackboard architecture was originally developed for speech recognition in the late seventies and early eighties and was quickly adopted as a flexible, powerful architecture for many AI systems **[131, 132,133]**.

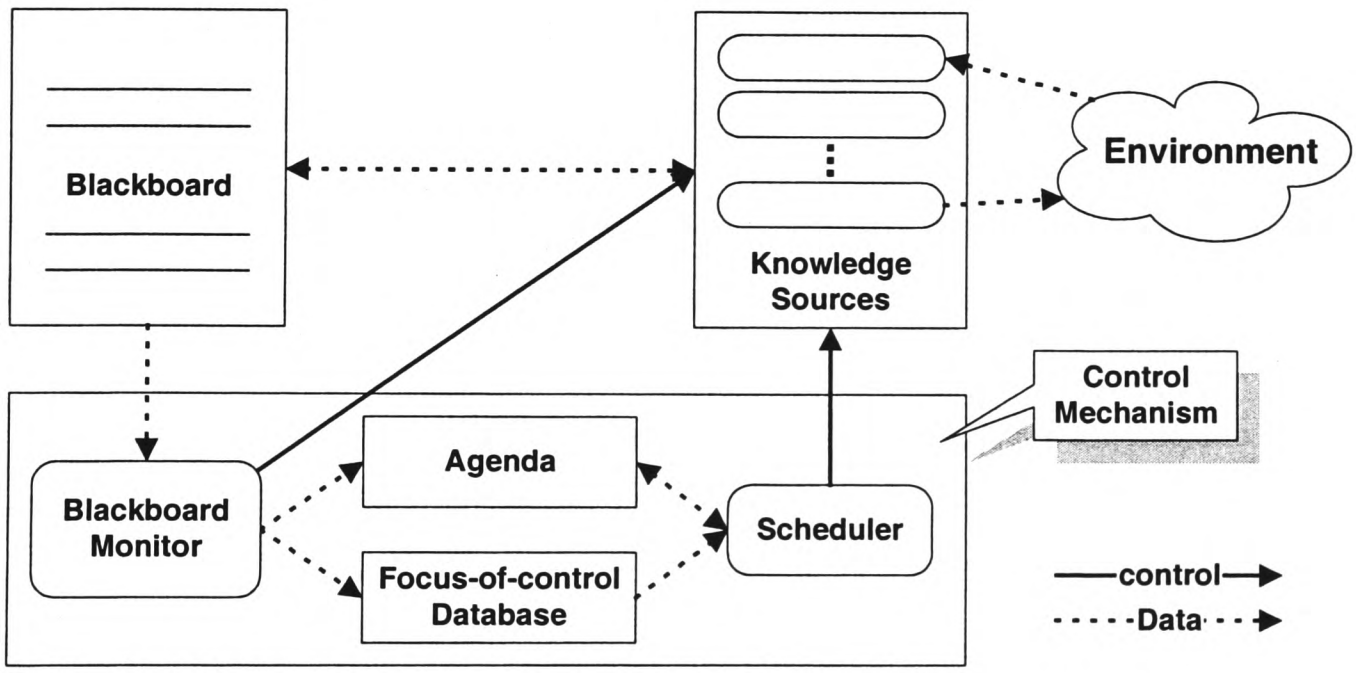


Figure 4-10: Blackboard architecture (extended from [133])

Blackboard systems contain three main components: the blackboard itself, a set of knowledge sources and a control mechanism (see Figure 4-10). The blackboard is a shared database in which each knowledge source can store and access data. *Knowledge sources* (KSs) represent the problem-solving knowledge of the system. They are computing entities used to analyse, add and modify data contained in the blackboard in order to achieve a goal (solve a problem). Key concepts of blackboard problem solving are that its activity should be *incremental* and *opportunistic*. Incremental means that solutions are reached by the augmentation and refinement of partial solutions contained in the blackboard. Each KS provides its solving knowledge and participate towards the building of a complete solution.

Opportunistic problem solving means that the system decides what action to take next (in order to reach its goals) according to the current situation.

In theory, KSs are independent and self-activating. They monitor the blackboard's state in order to decide if their activation is possible and appropriate. The activation of a KS should not involve other KSs and all communications between KSs are done through the blackboard. Therefore, blackboard systems could in theory be control-less systems in which each KS fend for itself. However, the reality of computing makes the existence of a control mechanism a necessity. Figure 4-10 illustrates a basic control mechanism. The blackboard identifies which KS could be activated and generates *knowledge source instantiations* (KSIs also know as knowledge source activation records) that it places inside the agenda. The scheduler rates the KSIs contained in the agenda to decide which one should be activated. The focus-of-control database is used to focus the systems activity on the most promising parts of the blackboard.

Partly because of these AI roots, it is widely used in cognitive MAS. It offers many advantages when applied to implement autonomous agents. The opportunistic triggering of KS offers a perfect mechanism for autonomous behaviours. Moreover, the blackboard itself represents an ideal communication medium between agents inside the system. Blackboard systems are sometimes viewed as a flexible "meta-architecture" than can be used to implement any other agent architecture. However, it suffers from the heavy control mechanism and can prove inadequate for real-time systems.

4.3.5.d Competitive tasks architecture

Whereas the subsumption architecture requires that the links between behavioural modules be defined and fixed during design, other models have been proposed that allow dynamic linkage of such modules at runtime **[134, 135]**, thereby increasing the flexibility and adaptability of agents.

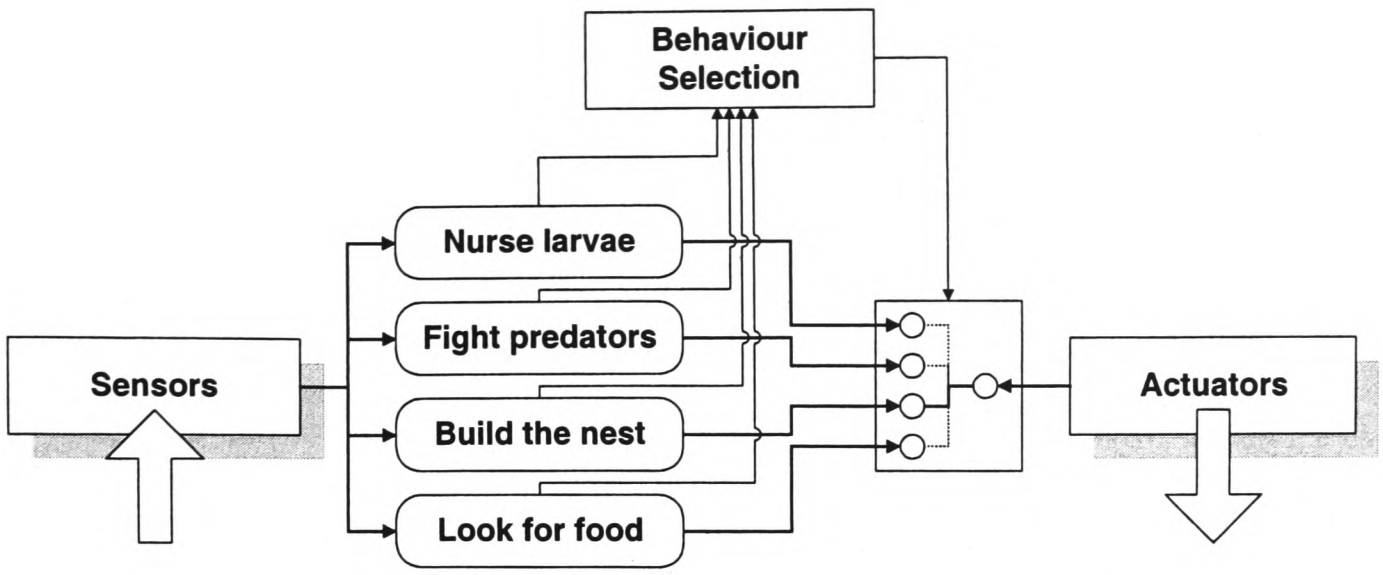


Figure 4-11: Competitive tasks architecture (ant example)

A particular example of competitive tasks architecture is one presented by Drogoul and Ferber with the MANTA simulator **[11,106]**. MANTA is a MAS used to simulate the activity of ant colonies. Individual ants used the competitive tasks architecture in order to dynamically adapt their behaviours to the situation. Figure 4-11 shows how the individual behavioural module (or task) of an ant can be made to compete for selection by a decision mechanism. The behaviour selection decides which task should be activated according to internal weights affected to each and also to the current situation. Feedback is provided through the environment and allows selection to act as a behaviour reinforcement (in the case of positive feedback) mechanism.

Selection of competing tasks allows for better flexibility in agent's behaviours. It is an efficient method to select between different high-level tasks that an agent can perform in given situations. For example, ants might prioritise the collection of food over nest building during shortage periods. Moreover this architecture allows pseudo-subsumption between low-level and high-level behaviours through allocation of significantly higher selection weights to low-level reflexes.

4.3.5.e Rule-based architecture

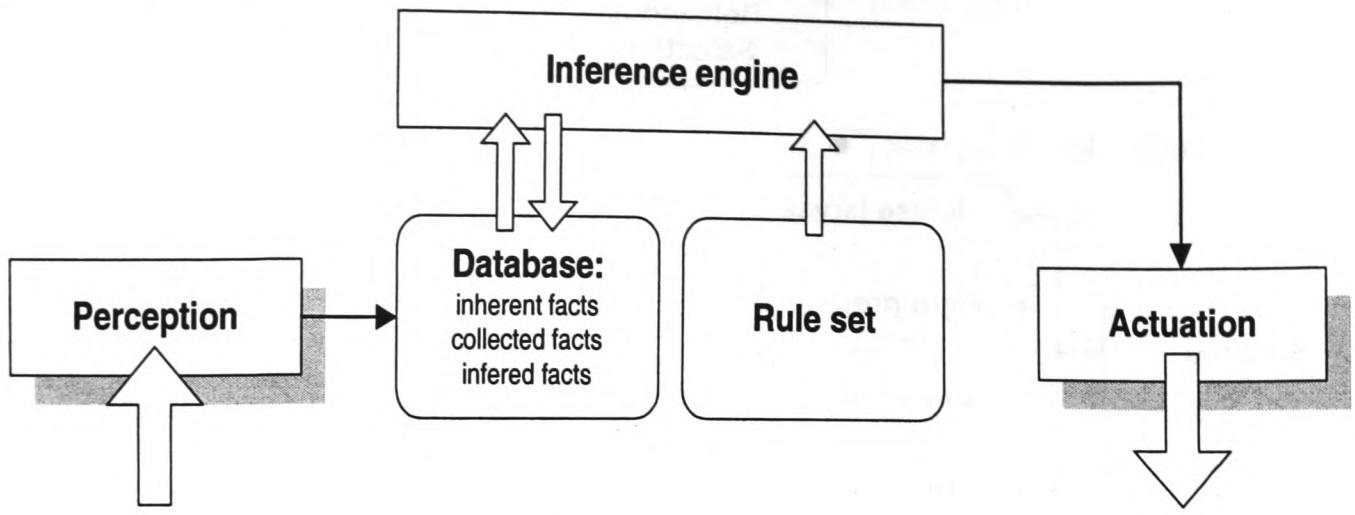


Figure 4-12: Rule-based architecture

Rule-based systems (also called production systems, inferencing systems or more generally expert systems) are another popular approach in AI that can be used to implement agents **[96]**. They have the interesting property that their logic is not pre-defined but that it depends on both external event and internal data. Rule-based systems are composed of three main elements (see Figure 4-12):

- A **database** holds all the facts known to the system that can be long-term persistent or volatile knowledge. Facts can represent collected information about the environment or knowledge generated by the inference engine.
- A **rule set** (or rule base) contains the long-term knowledge of the system encoded in the form of simple IF-THEN rules. The rule set if usually set at design time and represents rationalised knowledge about a given domain. Each production rule expresses one or more conditions and an associated list of actions. Actions can involve adding, modifying or deleting facts from the database, or executing agent tasks through an actuation mechanism.
- The **inference engine** is the "machinery" that allows a rule-based system to use the rule set with information inside the database in order to generate new facts, take actions and generally fulfils its function.

The rule-based approach is suited for problems of limited scopes, where it is possible to efficiently express the specific knowledge in terms of production rules. In such cases, it provides an elegant way of capturing problem solving skills and to apply them. Facts about an agent's environment are simply added to the database by a perception module and are immediately available to the inference engine to work upon. Adaptive, learning capabilities

can be added to rule-based agents by allowing them to modify their rule set according to relevant experiences.

However, this approach also has drawbacks. Whereas human beings can see conflicts between two proposed actions, the agent merely see them as two actions. When several rules apply to a given situation, the order of application can modify the outcome. Therefore there must be a high human involvement in checking the consistency of rule sets. Consistency problems also occur inside the database when modifying the rule set. In running systems, changing rules can cause previously inferred data to become invalid. Restarting the system to ensure consistent facts may destroy useful short-term knowledge. These problems are made more relevant in complex systems that require large numbers of rules.

4.3.5.f Beliefs-Desires-Intentions (BDI) architecture

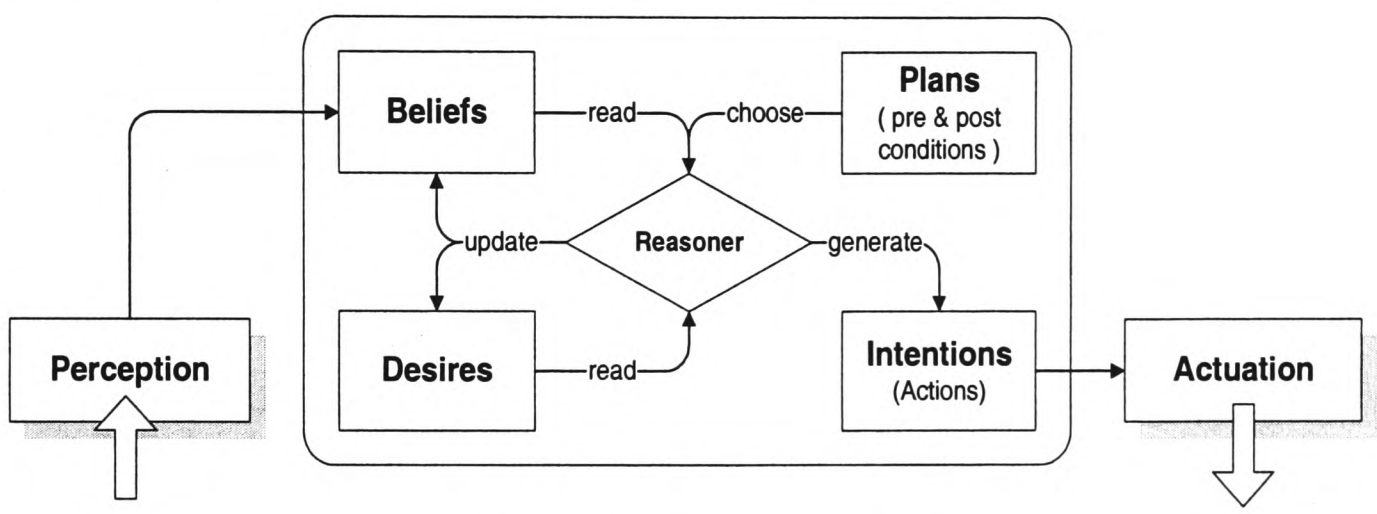


Figure 4-13: BDI architecture

The BDI agent architecture was first introduced by Georgeff and Lanski **[120]** and is now widely accepted as elegant, flexible and powerful **[136, 137, 138, 139, 140]**. From an internal point of view, a BDI agent in specified by three elements (see Figure 4-13):

- **Beliefs** describe the information about the environment and the internal mental state of the agent. Beliefs are maintained by the agent to reflect the current situation. Beliefs can represent collected data (from perception), deduced facts about the environment or internal mental states of the agent.

- **Desires** describe the goals that an agent may pursue during its life. They usually express a state that the agent or its environment should reach. Desires can be pre-defined or dynamically generated.

- **Plans** describe the course of actions that an agent may possibly employ to achieve its goals. It represents the skills available to agents. They are usually defined as a set of

pre- and post-conditions used to assess their applicability in given situations and the potential consequences of their application respectively.

A central **reasoner** uses these three elements to generate the agent's behaviour. Desires are used to justify activity. When some desires are not fulfilled, the reasoner looks for an adequate course of action in the plan database. If one if found, it is used to generate intentions that will be turns into actions by actuators.

BDI agents are popular for two reasons. Firstly, they provide an elegant decomposition scheme for motivated agency by separating the different elements of agent's rationality. Secondly, BDI agents are able to adapt their action to the situation more efficiently than other agents are. BDI agents can acquire the information about how best to achieve its goals during the plan execution. They are also focused on offering adequate response to situations when other planning agents often commit themselves too strongly to their plans. The BDI architecture therefore permits agents to always be in phase with their environment by strongly linking their action with the present situation represented by agents' beliefs.

4.3.5.g Multiagent architecture

It is possible to use a multiagent system as the internal architecture for agents. The resulting nested MAS have been illustrated earlier in Figure 4-1. The level of abstraction typically decreases as the level of nesting increases. Top level agents perform complex abstract tasks while low-level nested MAS handle the simple mechanics of underlying systems.

4.3.5.h Neural networks

Neural networks work on the metaphor of the brain by creating complex interconnection between identical entities called *formal neurones*. Each neurone is a basic input/output box, which determines its output according to its inputs using a transfer function.

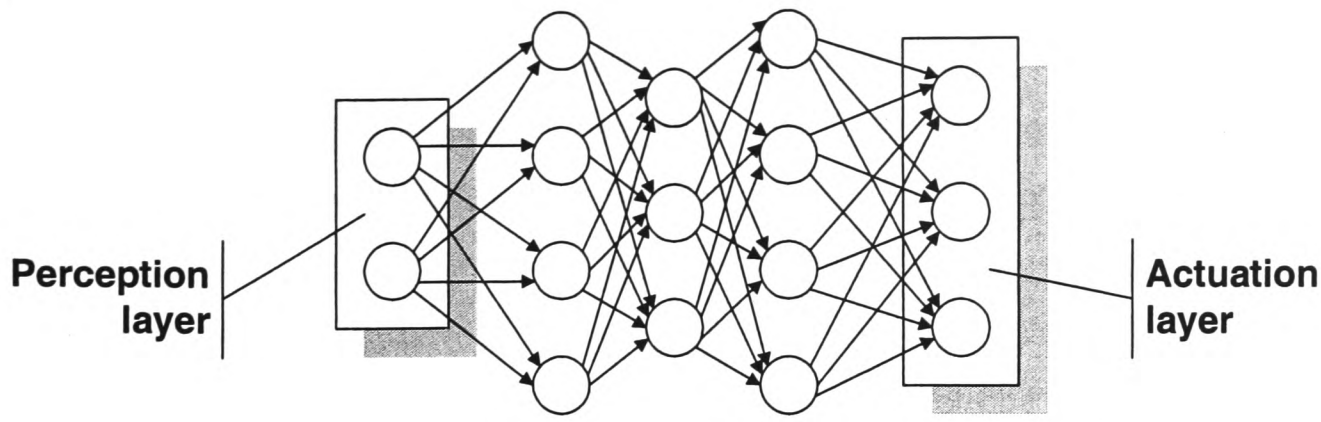


Figure 4-14: neural network architecture (a layered net)

The neural net approach can be used to implement reactive agents. Figure 4-14 shows how it can be done using neurones for perception, "*reasoning*" and actuation. The main advantage of the technique is simplicity as demonstrated by Beer and Chiel **[134]** who manage to create the most basic agent using only two neurones. Behaviours of such agents can be defined (it is called training the network) by modifying the weights of neurones' input. Adaptive weighting inside the network can be used as a behaviour reinforcement mechanism.

4.4 Interaction, Co-ordination and Co-operation

The power of MAS comes from the ability of autonomous agents to interact with one another in the pursuit of their goals. Only interactions allow a MAS to achieve more than the sum of its constituting agents. Interaction between autonomous agents comes in various forms; co-ordinated actions and co-operative actions being the most interesting. The concise Oxford dictionary offers the following definitions:

- Interaction is a "*reciprocal action or influence*".
- Co-ordination means "*working or acting together effectively*".
- Co-operation means "*working together to the same end*".
- Delegation is "*the act of committing (authority, power) to an agent or deputy*"

The later three terms describe the main types of interaction occurring inside MAS. **Co-ordination** defines interactions between agents that do not necessitate shared goals or negotiation. These are typically encountered in non-deliberative agents. For example, the activity of collision avoidance between moving agents represents a co-ordinated behaviour. Two agents on a collision course will alter their trajectory in order to avoid colliding. However, decisions are taken according to their perception of the situation and do not involve complex negotiation between agents. Moreover, although both agents have an

identical goal (in this case), it is not shared between them. Agent-A aims at avoiding collision from his perspective and Agent-B does the same from his. This is not equivalent to aiming for a shared goal: avoiding each other. One advantage of co-ordination is its localised applicability. No communication, sharing of intentions or negotiation is necessary to carry out co-ordinated actions. Instead it relies on perception of the environment and the local agent's skills to solve local problems. This approach has proved its potential to cope with fairly complex problems and is computationally inexpensive. However, this is only true if interacting agents have compatible behaviours. Imagine agent A and B heading directly towards each other. Agent-A avoids collision by moving to his left on a parallel course. Agent-B's behaviour is to do a similar manoeuvre but to its right. BANG! These two simple behaviours could prove incompatible and a head-on collision might not be avoided!

Co-ordination seduces by its simplicity and reduced computational requirements. However, it has just been shown that it requires that interacting agents apply compatible behaviours. **Co-operation** between agents represents interaction with a shared goal or negotiated actions and is typical of deliberative agency. It allows agents to address the issue of incompatible behaviours by introducing negotiation before action. Co-operative actions involve a heavy communication and reasoning overhead. Co-operating agents use peer-to-peer communication to exchange their goals and intentions. Each protagonist assesses the potential consequences of others actions on its goals and possibly the impact of its actions on other agents goal. If incompatibilities arise, alternative courses of action are proposed and negotiated among the participating agents. This process goes on until a plan acceptable by all is reached and agreed upon. In some cases however, such a unanimous decision is not possible and an alternative decision making process must be used to resolve the deadlock. This might involve prioritising one agent's welfare to the detriment of others or requiring assistance from more competent agents or the user. Co-operation relies heavily on communication for goal sharing and negotiation. It requires complex reasoning capabilities that only deliberative agents provide at the expense of performance.

The final interaction type occurring inside MAS involve a defining property of agency: **delegation**. The usefulness of agents comes from their ability to carry out one or several specific tasks on behalf of others. It has just been discussed that agents can encounter problems that their limited skills can not crack. In such cases, agents can attempt to delegate the challenging task to more qualified agents or even request user assistance.

4.4.1 Architectures for interaction

Interactions among agents create the driving force of MAS. The type of interactions supported defines a large proportion of MAS capabilities and is therefore a crucial part of MAS design. In **[141]**, Patriti defines three metaphors used for modelling agent interaction in self-organising systems.

- The **biological** metaphor uses agents mimicking behaviours of animals inside their eco-system. This approach usually involves reactive agents, possibly marking their environment and has previously been discussed in 4.3.3.a. In such systems, agents have predefined tasks triggered by perception of their environment.
- The **psychological** metaphor uses agents that reproduce the internal behaviour of conversing humans. This approach considers communications as actions and is based on the speech act theory. Each speech act is classified, permitting any agent to understand any other, providing they share common semantics (or ontology). The popular KQML is based on this theory and provides great flexibility in agent interactions.

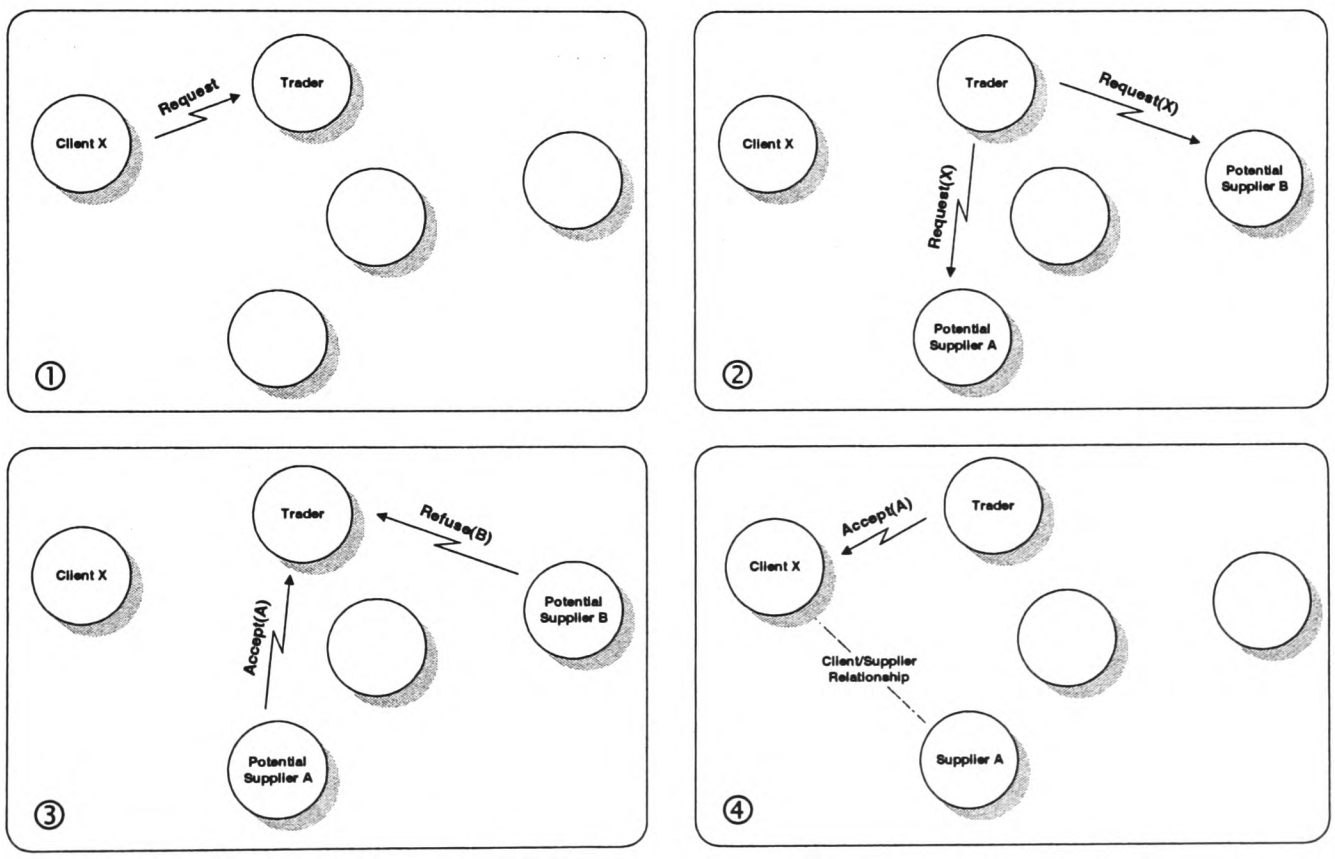


Figure 4-15: Trading approach to task allocation.

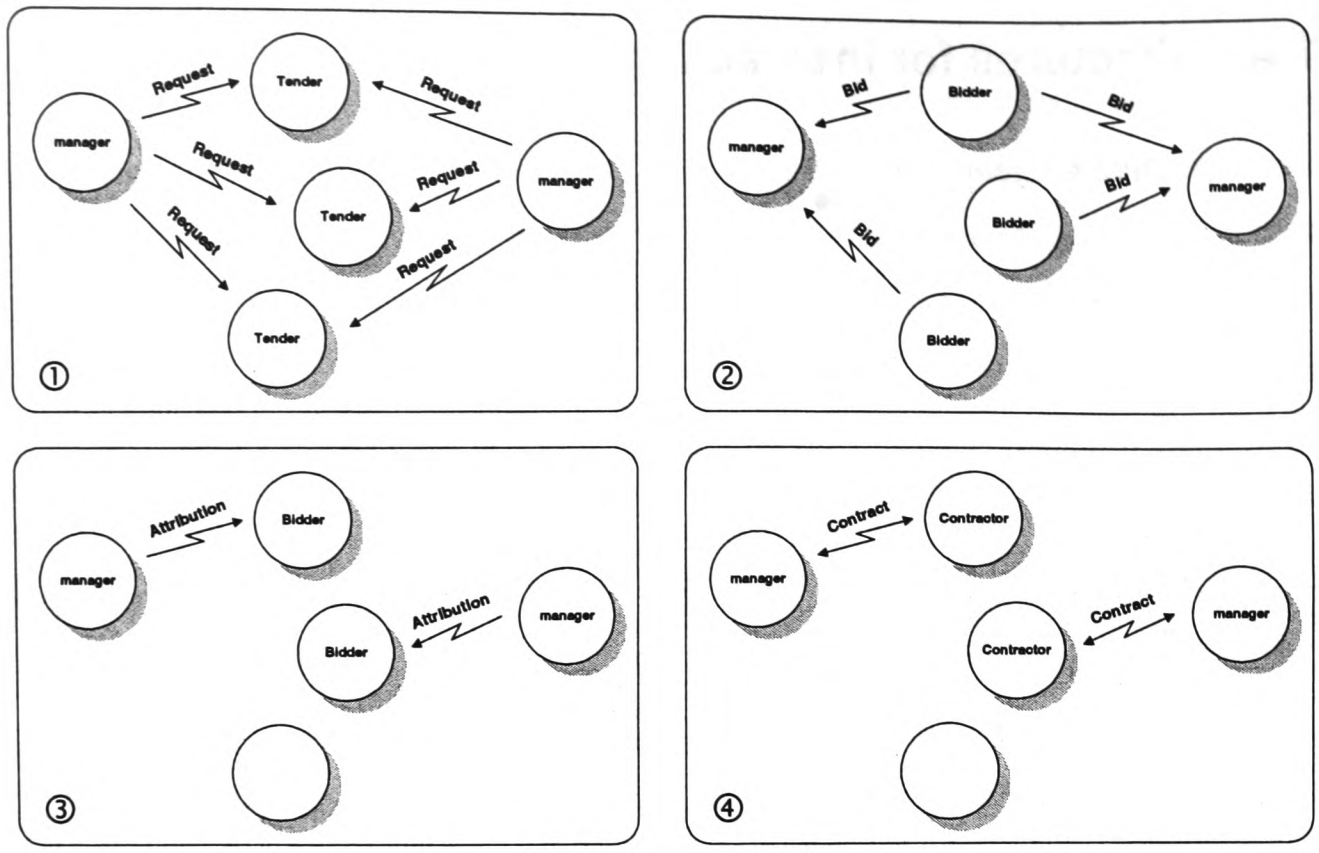


Figure 4-16: Contract net approach to task allocation.

- The **social** metaphor is very popular for task allocation inside MAS. It uses agents reproducing the behaviours of human beings when acting inside a group. The game theory belongs to this metaphor. Task allocation is done through choice matrices between each agent and all others. The cost or profit of task performed by each agent is calculated and stored in the matrix and used to allocate tasks. Wooldridge shows that the determination of cost function can be problematic **[142]**. Other social metaphors used in agent task allocation are subordination, trading and contract nets. *Subordination* involves a "boss" agent imposing tasks on subordinate agents and is used in a fixed organisation. More complex organisations are egalitarian, and use a co-ordinated selection process to allocate tasks. The *trading* approach (see Figure 4-15) uses trader agents, who receive task requests from other agents, forward them to appropriate agents and attribute tasks according to the proposition received. The trader keeps a database of agents' skills and uses it to find potential suppliers. He also has final word on the attribution when several potential suppliers accept a task. The *contract net* approach (see Figure 4-16) is even less hierarchical and eliminates the need for dedicated traders. Any agent that requires a task performed becomes a *manager*. Managers broadcast requests to all other agents (tenders) and await bids for a given period of time. Agents interested in performing a requested task can bid for it. At the end of the bidding period, the manager decides who can best perform the task (cost evaluation) and attributes it to the future contractor. A variation of contract net is

the *acquaintance net* which introduces a degree of fidelity between client and contractor by allowing direct delegation requests.

In the case of problem solving using agent-based systems, it is possible to categorise interaction architectures (or protocols) between *satisfactory* and *optimal*. A protocol is optimal if it searches for the best solution in the solution space. It does not means that it finds the best existing solution but that it returns the best solutions of those it examined. A protocol is satisfactory if it return any possible solution without evaluating its relative quality. Because of the locality of interactions, multiagent interaction protocols are mostly satisfactory.

4.4.2 Emergent behaviour

Emergence **[10]** occurs "*when a complex, orderly pattern arises from interactions among simpler objects. Building blocks at one level combine into new building blocks at a higher level, when agents organise into larger structures. At each level, a new emergent structure would form and engage in new emergent behaviours*" **[143]**.

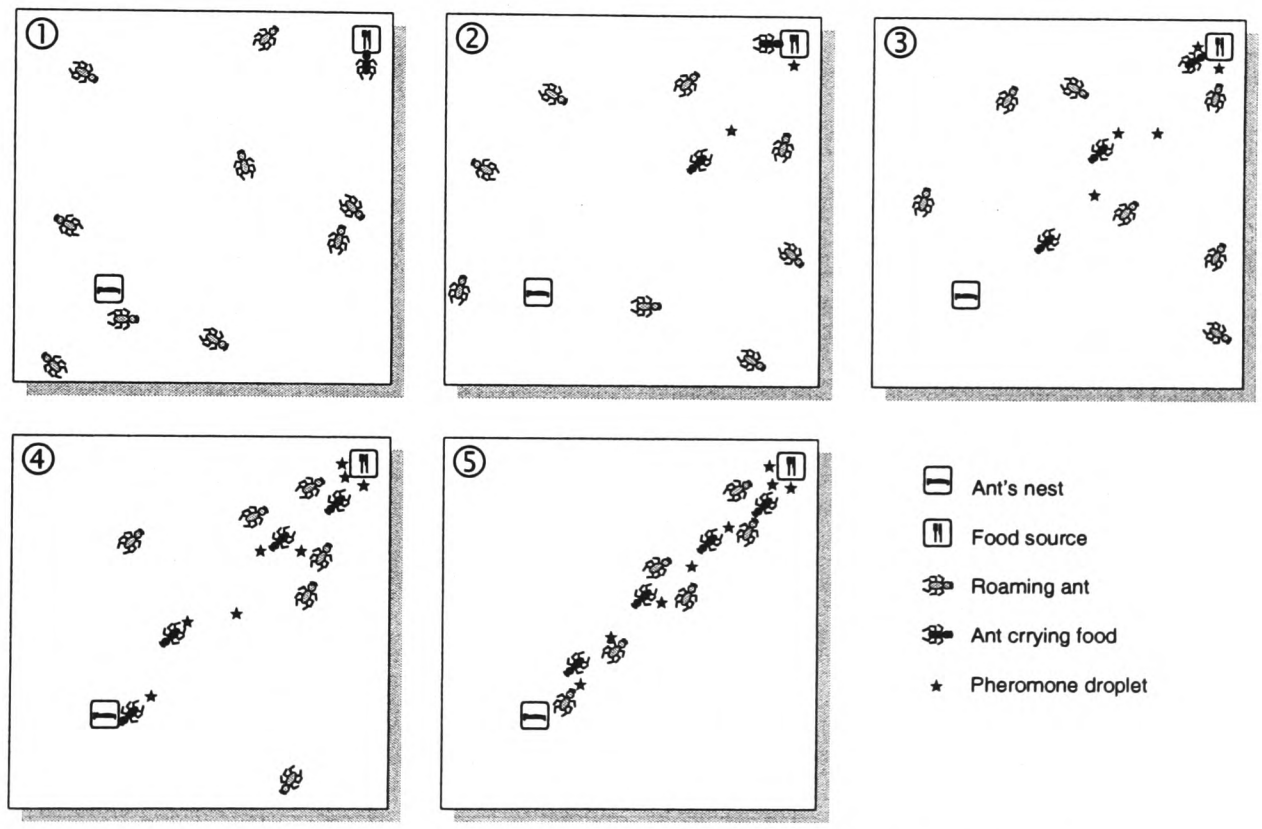


Figure 4-17: Emergence of complex behaviour in ant colony

The concept of behaviour emergence is crucial to MAS and particularly to non-deliberative agency. It has been discussed informally in section 4.2.3 and section 4.3.3.a, when using the classic insect-colony analogy. Let's use this same analogy again to

demonstrate the power of emergent behaviours. Consider once again these simple behavioural rules defining ant-agent's behaviour:

```
IF (carry nothing) THEN (roam AND look for food)
IF (find food) THEN (pickup food AND drop pheromone)
IF (carry food) THEN (go to nest AND drop pheromone)
IF (in nest AND carry food) THEN (drop the food)
IF (hungry AND tired) THEN (go to nest)
```

A simple scenario is to place an ant nest inside an environment containing a unique food source and observe the behaviour of the colony (see Figure 4-17). Until the food source is discovered, all ants roam the land randomly in search for food. When the first ant finds the food (see Figure 4-17①) it drops some pheromone and starts carrying some food back to the nest. Quickly the pheromone sent will make other ants converge to the food source (see Figure 4-17②③ and ④). Every time food is picked up, pheromone is dropped therefore strengthening the attraction to the other ants and preventing the scent's natural decay. Because ants carrying food drop pheromone at regular interval on their way back to the nest, a path to the food source is marked and maintained as long as food is brought back form it. In a short period of time, the colony has organised itself and focused it attention to a single bi-directional path to the unique food source (see Figure 4-17⑤). This complex behaviour did not need to be defined in a lengthy rule-set. Instead, it naturally emerged from the simple interactions between individual ants, thereby proving the power of emergence in MAS.

Such emergent behaviours are obtained by supplying agents with localised knowledge and skills and rely on agent interaction. This approach contrast greatly with the *top-down* centralised control mechanisms used in conventional systems. Emergence is highly seductive because it allows for complex behaviours to be achieved in a flexible, modular and elegant manner. Most importantly, it achieves global coherence by taking advantage (and sometimes revealing) natural properties of distributed systems not used by conventional modelling techniques.

4.5 Agent Communication Language (ACL)

It has been made clear that communication among agents is a critical part of their modelling power as it is the main medium for useful interactions. Mayfield et al. argue that "*languages that facilitate high-level communication are [...] an essential component of intelligent software architecture*" **[103]**. The communication should not be mistaken for lower level communication protocols. On one hand, protocols such as TCP/IP, FTP or HTTP, provide mechanism for "physically" transport bit streams from one point to another.

On the other hand, communication languages provide methods to capture additional information about messages being sent between agents. ACLs might use communication protocols as their underlying transport mechanism and represent higher-level communication methods.

Finin, Labrou and Mayfield have identified the requirements for ACLs **[103, 144, 145, 146]**. They divide them in seven categories:

- **Form**: A good ACL should be declarative, syntactically simple and concise. Because even basic agents need to communicate it should be easy to parse and generate. Messages must be expressible in bit stream form compatible with the physical transport medium.

- **Content**: ACL should be layered in order to fit various systems. In particular, a distinction should exist between the communication language, which convey communication acts and the content language, which carries facts about the domain. The language should commit to a well-defined set of communication primitives (acts) but also allow extensions to be added.

- **Semantics**: Agents from different domains might understand different semantics specific to their domain. Semantics provide the conceptual framework to understand the content of messages. A good ACL must provide a mechanism to define the semantic used inside messages.

- **Implementation**: Implementation should be efficient in terms of speed and bandwidth requirements. The language should supports partial implementation because simple agents may only need a subset of the existing communication acts.

- **Networking**: ACLs should fit well within modern networking technology. It should support various connection types: point to point, multicast and broadcast both in synchronous and asynchronous mode.

- **Environment**: MAS are potentially distributed, heterogeneous system. A good ACL supports interoperability between potential computing environments.

- **Reliability**: ACL should support reliable communication. It should be robust to malformed and inappropriate messages and be able to generate warnings and errors when such situations arise.

Based on these requirement Finin et al created the Knowledge Query and Manipulation Language (KQML) which is becoming a de-facto standard in agent communication. KQML is presented in more detail in 4.7.1. It should be noted that the speech act theory used as a basis for KQML also has drawback **[147]** and alternatives exists.

Agent frameworks and architectures proposed to this day usually include specifications for an ACL. Both OAA and FIPA are in this case and are discussed in 4.7.1. Other specific languages have emerged from various agent-oriented projects. For example, **AgenTalk** is a language aimed at MAS using contract net type of interactions **[148]** and is used inside AgentSheets, a commercial agent development toolkit.

4.6 Issues in multiagent systems

The theoretical and experimental foundations of agent-based systems are actively studied and are increasingly well understood. More practical aspects of developing MAS are also under the scrutiny of the research community **[149,150]**. Some of the important issues to consider when dealing with agents are now presented.

4.6.1 Agent granularity

The granularity of agents used inside MAS is critical when attempting to benefit from the new agent paradigm. It should be chosen so that each agent embodies an entity, which can be isolated from others and possessing a clear function within the system. Great variation can exist from system to system but care must be taken to ensure consistent granularity between agents whose interactions generate the desired emergent behaviour. The importance of granularity regarding the distribution of agents was discussed in section 4.3.2.b. In **[149]**, Wooldridge also warns against the risks of applying the agent paradigm at the wrong level of granularity.

Using coarse agents, a MAS may contains too few agents to really reap the benefits of agency. In such a case the global behaviour of the system no longer emerges from the interactions between single function entities, thereby missing the main advantage of the agent paradigm. Noticeable exceptions to this are wrapper agents used to agentify legacy system.

Using fine-grained agents, a MAS may contain too many agents, which exposes two weaknesses of agency. Firstly, numerous agents working inside a MAS potentially create

high communication overhead (see section 4.6.2). Secondly, as the number of agents increase so does the predictability of the system, whose emergent functionality is akin to chaos.

4.6.2 Communication load

Peer-to-peer communication allows interaction between agents. In turn, it is the autonomous interaction between agents that makes multiagent systems work. In particular, software agents completely rely on communication for environment perception and task delegation. Without inter-agent communication there could be no emergent behaviour inside software based MAS.

Each agent is capable of initiating peer-to-peer communication with others inside the model it inhabits. The potential for exponential increase of communication load is clear as agents are added to the system. Indeed, with **n** autonomous agents inside the system, there exist **n(n-1)/2** potential peer-to-peer communication channels (bi-directional). Although individual communications are not computationally very expensive, high numbers of them can create a heavy workload for the system. A poorly designed MAS inhabited by numerous agents (see section 4.6.1) could spend most of its time handling inter-agent communication instead of carrying out primary agent functions. It is thereby important to keep a tight control over the global communication load created by agents. It should be noted that in the case of distributed agents, the issue of communication latency adds to the necessity of minimising overall communication load.

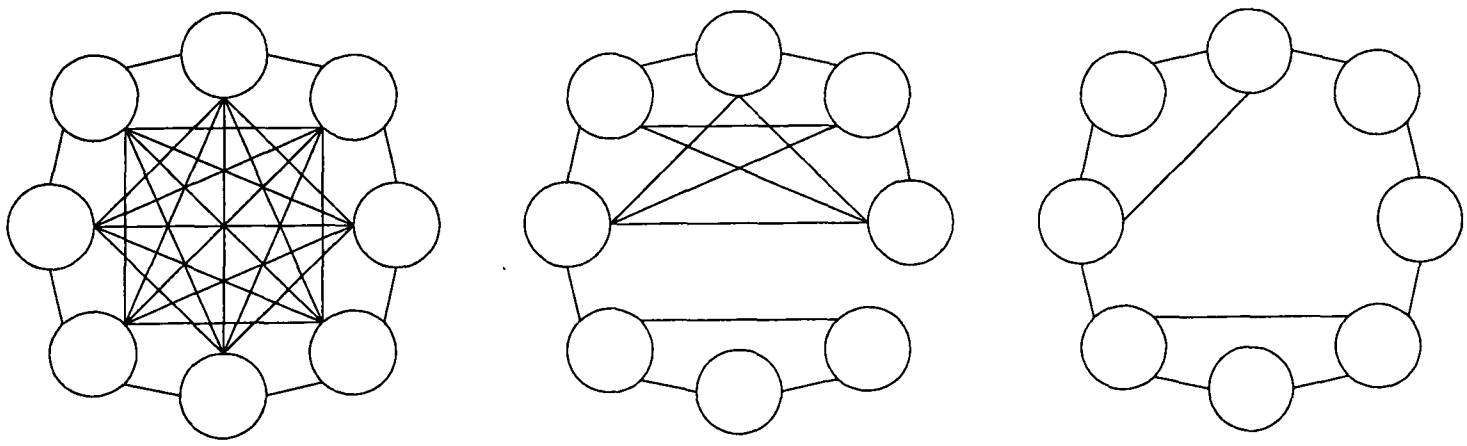


Figure 4-18: Clustering reduces potential communication channels

However, the locality of agent's activity **[104]** provides a natural way of controlling the global communication load by limiting the range of communication of each agent. Based on the agent's locality it is possible to create clusters of agents, which greatly reduce the overall

communication load as illustrated in Figure 4-18. Such clusters represent islands of communication and help containing communication in local areas of the design, thereby reducing global load. Despite the necessity to provide inter-cluster communication channel, clustering provide significant reduction in potential communication channels.

4.6.3 Conflict resolution

"By reason of their autonomy, that is, their capacity to determine their own behaviour, agents are led into situation where their interests may be contradictory. They are then, objectively, in conflict situation." **[98]**.

The resolution of conflicts is vital to ensure a MAS maintains its usefulness. Deliberative agents mostly attempt to resolve conflicts by building complex co-operation relationships which result in shared plans **[151, 152, 153]**. Non-deliberative agents generally deal with conflict in much simpler ways. Behaviour degradation is often the only option to eliminate conflicts. Situation may arise where reactive or situated agents are unable to resolve conflicts, which results in the system locking. A deadlock happens when the system freezes and can no longer run. A livelock, as defined in **[154]**, occurs when the system enters a never-ending chain of event and can no longer stop (see section 6.2.1.a).

4.6.4 Agent serialisation

According to Muller, agents should represent things rather then functions **[153]** in order to provide their full benefit. Agentified things become runtime entities empowered with all the functionality described in 4.2.2. In particular a thing-agent will build an internal representation of its environment in order to perform its function. For example, a mobile robot modelled by mean of agent uses its perception to construct an internal image of its surrounding that allows it to decide how to move into it. This dynamically generated map of the environment is crucial to the efficient operation of the agent. It holds vital knowledge accumulated by the agent over its life in the system and erasing it would represent an important lose for the agent.

Because they are runtime entities, the internal state of software agents can be difficult to preserve in an efficient format. Most object-oriented programming languages used to implement agents offer mechanisms to *serialise* objects into storable digital files. However, such direct binary filing of running agents involve heavy information redundancy because the full execution context must be saved along with the object themselves. Moreover, most of

the object serialisation schemes are not adapted to agent-based software in which inter-agent exchanges play a crucial role in the system's operation.

No agent framework currently exist that offers fully functional serialisation of autonomous agents possessing internal world representation. The system designer is therefore responsible for planning and implementing the serialisation mechanism during the conception of the multiagent system. In particular, much attention must be paid when deciding what portions of the world-view should be preserved and what portions should be discarded based on their nature. In all cases, a validation mechanism must be created that checks the restored internal state of the agents against the current situation of the system.

4.7 Survey of Existing Systems

This section provides a survey of significant existing implementations of both multiagent framework and applications. It does not pretend to be an exhaustive list of all developed MAS but instead aims at presenting the implementations that are most notable in the field of MAS and/or relevant to this thesis.

4.7.1 Agent frameworks and tools

With the fast emergence of agent-based paradigm as an alternative in software development, various projects exist that aims at addressing the need for industrial-strength architectures, frameworks and development tools. The number of available tools for creating agent-based systems is substantial and this section only presents the most significant. In particular, it does not present the numerous implementation tools accessible to developers such as JAT **[155]**, Swarm **[156]**, COALA **[157]** and others.

Agent frameworks and tools: ① Distributed Objects, CORBA and DCOM

The adoption of a networked computing model is leading to a greatly increased reliance on distributed sites for both data and processing. Object-oriented programming (OOP) languages have long allowed for creating monolithic application out of many object building blocks. Distributed object technologies such as *Common Object Request Broker Architecture* (CORBA) **[158]** by OMG and *Distributed Component Object Model* (DCOM) **[159]** by Microsoft® permit application's components to spread across multiple machines. These technologies provide a *software bus* between distributed objects, which permit transparent invocation of class variable and methods. This means that through such software buses,

application gain access to information transparently, without having to know what software or hardware platform it resides on or where it is located on an enterprises' network.

Even though these infrastructures are based on a client/server architecture, their emergence is a signal that computing is getting increasingly decentralised and distributed. This decentralisation offers a natural environment for agent-based applications. They can be used as a technological substrate on which to create agent-based architecture. In **[160]**, Nwana points at work already achieved to build an agent layer on top of CORBA compliant systems.

Agent frameworks and too ls: ② KSE, the Knowledge-Sharing Effort (KIF, ontolingua, KQML)

The **KSE** is an initiative to develop technical infrastructure to support knowledge sharing among systems. It is organised on three working groups addressing complementary problems identified in knowledge representation technology. Three main contributions have been produced by the KSE. The Knowledge Interchange Format, **KIF**, is a common language for expressing the content of knowledge base **[161]**. KIF can be used to support translation from one content language to another or as a common content language between agent using different internal representations. Practically, sharing knowledge requires more than a formalism (KIF) and a communication language to work. Every knowledge base relies on some conceptualisation (ontology) of the world that is used to store symbolic representation on things. KSE addresses the need for common ontologies through **ontolingua [162]**. It has constructed ontologies (using KIF) for various domains that can be used off-the-shelf by communicating applications. Finally, the KSE has produced a communication language for knowledge sharing: **KQML**, the Knowledge Query and Manipulation Language **[144, 145, 146]**. KQML is a layered language, offering complete separation between communication data (sender, receiver, message type) and message content. It also allows for defining the ontology used inside the content layer thereby allowing heterogeneous application to communicate meaningfully. Because of these qualities and because it is easy to parse and generate, KQML is becoming a de facto standard in the agent community.

Agent frameworks and too ls: ③ OAA, the Open Agent Architecture

"*The Open Agent Architecture provides a framework for the construction of distributed software systems, which facilitates the use of co-operative task completion by flexible, dynamic configurations of autonomous agents*" **[163]**. OAA's agent library, which provides the necessary infrastructure for constructing agent-based systems, is available in several programming languages (Prolog, C, C++, Java, Lisp, Visual Basic and Delphi). They provide

both intra-agent and inter-agent infrastructure; that is, mechanisms for supporting the internal structure of agents and mechanisms for inter-agent interoperations. OAA supplies an inter-agent communication language similar to KQML with a content layer based on PROLOG similar to KIF. Peer-to-peer and broadcast exchanges are possible through *facilitator* agents that hide the complexity of messaging to its *client* agents. Facilitators also offer complex services such as accepting compound goals, dividing them into sub-goals and assigning them to different agents in the system. OAA triggers provide a general mechanism for requesting that some action be taken when some set of conditions is met. They can be of four types: communication, data, task and time. OAA is a recent framework that offers powerful functions in a nicely wrapped package that make MAS creation a simpler task.

Agent frameworks and too ls: ④ CoABS, Control of Agent-Based Systems

CoABS is a large project sponsored by the American army (through DARPA) to investigate the use of agent-based software for operational co-ordination **[164]**. CoABS aims at creating software agent that cut by a factor of 10 the time spent manipulating information. Thereby allowing war-fighting personnel to spend more time focussing on their mission and less time manipulating information systems. The motivation behind this effort is that "*large-scale, co-operative teams, comprised of interacting agents [...], could offer new capabilities that are now beyond the realm of software designers. An infrastructure that could provide these capabilities allow software developers to design smaller pieces of code that would primarily function on solving problems via interaction with each other, rather than by trying to duplicate functions provided by others*" [extract from http://coabs.globalinfotek.com].

CoABS investigate several aspects of agent-based control:

- Co-operative control strategies
- Algorithm, system designs, policies, and methods for behaviour control
- Computer system architecture
- Related technologies

Agent frameworks and too ls: ⑤ FIPA, Foundation for Intelligent Physical Agent

The Foundation for Intelligent Physical Agents (FIPA) is a non-profit association registered in Geneva, Switzerland. FIPA's purpose is to promote the success of emerging agent-based applications, services and equipment. This goal is pursued by making available in a timely manner, internationally agreed specifications that maximise interoperability

across agent-based applications, services and equipment. This is realised through the open international collaboration of member organisations, which are companies and universities active in the agent field. FIPA intends to make the results of its activities available to all interested parties and to contribute the results of its activities to appropriate formal standards bodies **[165]**.

FIPA can be viewed as a similar effort to CORBA in the distributed object arena, but addressing agent-based applications. The FIPA standards provide:

- a commonly agreed means by which agents can communicate with each other so they can exchange information, negotiate for services, or delegate tasks
- facilities whereby agents can locate each other (i.e. directory facilities)
- an environment which is secure and trusted where agents can operate and exchange confidential messages
- a unique way of identifying other agents (i.e. globally unique names)
- a means of accessing non-agent and legacy systems, if necessary
- a means of interacting with users
- a means of migrating from one platform to another, if necessary.

Agent frameworks and too ls: **© KAoS, Knowledgeable Agent-oriented System**

KaoS is another effort in creating an open distributed agent architecture **[166]**. An important characteristic of this architecture is that it pioneers conversation policies as protocols to represent the interaction dynamics of agents' interactions **[167]**. It includes the notion of *agent domains*, which represent bounded areas of agent activity, and uses *domain manager* agents to control their input/output with other agent domains. KaoS also uses concepts such as *proxies* and *mediators*, which mediate the inter-communication among KaoS agents from different object models and the inter-communication with on-KaoS agents respectively.

4.7.2 Multiagent applications

The number of multiagent systems of interest is considerable and growing at an almost exponential rate. The thesis will therefore only present MAS applications in the engineering domain and particularly in engineering design. Other interesting applications of MAS for

simulation **[11,106]**, air traffic control **[168]**, information retrieval and filtering **[96]** show the great potential of agent-based systems **[8]**. The following section describes some recent applications:

Multiagent applications: ① C a ble Harness-Design

In **[169]**, Park et al. present, **First-link**, "*an approach to providing computational support for concurrent design [...] in the context of an industrial cable harness design problem*". Three perspectives need to be considered while designing complex cable harness:

- Electrical properties embody the primary function of the harness.
- Geometry defines the space in which the harness resides.
- Configuration defines the harness as an assembly of elements (wires and connectors).

Autonomous agents that deal with these different aspects of harness design are used to assist the designer. A *cable editor* allows the designer to test various configurations rapidly by automatically handling the details and keeping tracks of constraints. A *part selector* generates part lists based on current configuration and constraints. A *free space manager* generates models of the available space for cable routing. A fourth agent called *environment editor* generates geometric representation of the environment the harness must fit in. Autonomous agent activity provides modification propagation between agents and ensures that modification made using one perspective does not compromise others automatically. This allows designer to test different configurations more easily as fast feedback is provided on the impact of changes. First-link is a great example of the potential of agent-based design system in collaborative projects. **Next-link** is a continuation of first-link aimed at testing co-ordination techniques between design agents **[170]**.

Multiagent applications: ② C ollaborative design in assembly

Mori and Cutkosky present an agent-enabled CAD system in which engineering design agents interact with each other, exchange design information and keep track of state information to assist with collaborative design **[171]**. The agents are wrappers for the commercial CAD product AutoCAD® 14 and are reactive in the sense that they keep track and react to changes in the design. Agents attempt to eliminate conflict between parts destined to be assembled through co-ordinated actions. Design agents are rule-based entities and might suffer from limitations inherent to rule-based system (see section 4.3.5.e). In spite of this, this approach is interesting because design agents are allowed to make autonomous

modifications to the proposed design and possibly lead the human designer in direction that would not have been explored otherwise.

Multiagent applications: ③ A A RIA manufacturing scheduler

AARIA (Autonomous Agent for Rock Island Arsenal) is an industrial-strength agent-based scheduling architecture being developed for an American army manufacturing facility. The needs-driven approach means that the system must respond the domain's specific requirement but also support the true agent capabilities. The infrastructure of the system allows true broadcast, multithreaded agent activity, agent migration and multi-platform instantiation **[172, 173]**. The agents programmed in Objective-C and running on a network of Pentium-based computers under PDO (Portable Distributed Objects), actively represents each step on the ladder of manufacturing a part. AARIA maps agents primarily onto manufacturing entities such as parts, machines, and operations. Parunak et al. see "*this mapping as a natural way to provide empowerment on the shop floor, and provides a rich set of interactions that can support [most required functionality]*" **[173]**. This agent-based scheduler takes advantage of agent technology to offer increased flexibility and adaptability in manufacturing. It represents a good example of real industrial application of MAS in a manufacturing industry. Note that part agents do not represent physical parts but part types. A part agent maintains the information about a part type, owns its inventory, keeps its production history and can forecast future production based on this information.

④ *Multiagent application s:* DFM via Agent Interaction

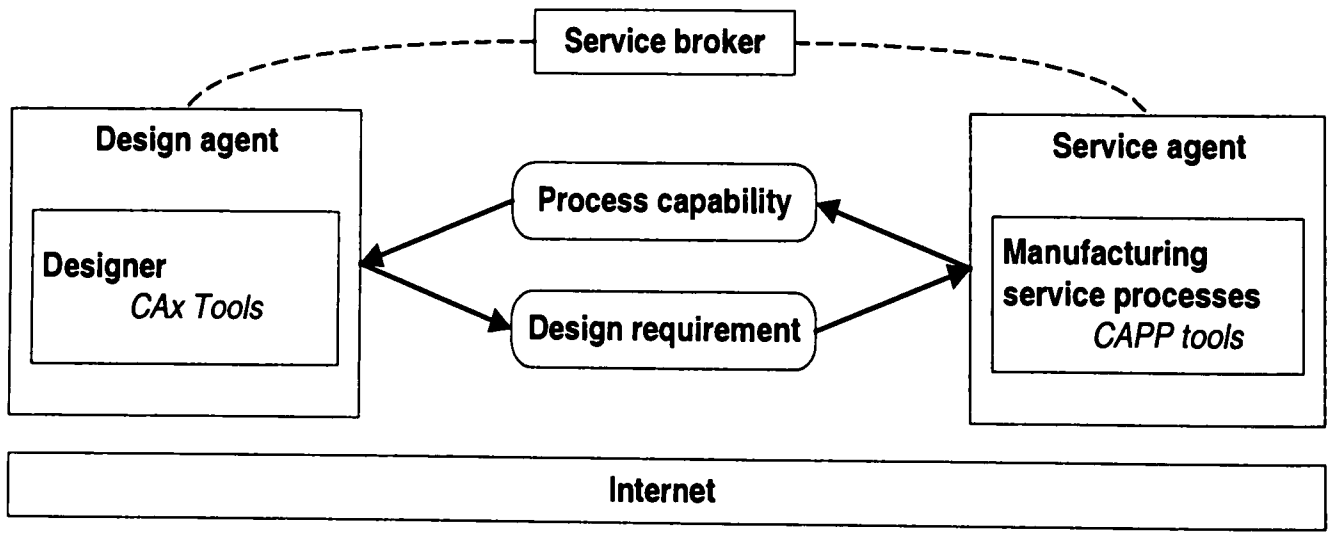


Figure 4-19: SANDIA's DFM agent architecture (from [124])

Sandia National Laboratories propose a DFM "*approach for making the capability of manufacturing process manifest to designers starting with the earliest stages of geometry specification*" **[124]**. The suggested architecture relies on the encapsulation of designer and

manufacturing services with software agents (see agent wrapping in section 4.3.4.b). The agent's abstraction is used "*as a mechanism for encapsulating and exchanging distributed knowledge and functionality*". Basically SANDIA adds agent wrappers around systems used each stage of the manufacturing process (starting with the design) to provide designers with results of incremental process planning and simulation at any time. Distributed design agents and manufacturing service agents can communicate using KQML messages containing STEP-EXPRESS data. Design agents express design requirements while service agents return process capabilities (see Figure 4-19). This approach allows fast an efficient feedback to be obtained by designers and supports highly distributed system close to the concept of virtual factory. At this time, it allows for an efficient sharing of information between design and manufacturing. Agent wrappers permit design requirements and process capabilities to be exchanged transparently between heterogeneous systems, thereby helping bridging the gap between CAD and CAPP. Up till now, they do not provide any active assistance to the designer concerning the decisions to be made.

Multiagent applications: ⑤ **M A DEsmart, agent support for design**

MADEsmart is a MAS developed by Boeing Corp. presented in **[125, 126]**. It is an "*integrated multiagent based prototype to help reduce the design/analysis cycle problem for the aircraft parts by, among other things, providing a manufacturability analysis of the designs*" **[126]**. MADEsmart is interesting because it combines various agent technologies inside a working prototype. Its agents are built using blackboard architecture, and communicate in KQML. It contains new autonomous agents for tasks such as information retrieval, task execution and project management but also wrapper agent around legacy systems used by Boeing (ICAD/CATIA and CASTADE). Interface agents are also used to allow user interactions with the MAS during design of mechanical parts. Execution agents encapsulate skills previously held by human experts (design, analysis, or optimisation) and often interact with appropriate human experts to accomplish their goals. MADEsmart enhance manufacturability of designs in two steps. Firstly, it provides appropriate manufacturing standards to the designers early in the design process (through efficient agent-based information retrieval). Secondly, it provides a direct manufacturability analysis of the designs by generating and analysing prototypical process plans for the designs (using the variant method). Using the variant method is efficient in Boeing's case as they rarely redesign parts from scratch but modify existing designs instead.

Multiagent applications: **© A B CDE, integrating design and shop-floor control**

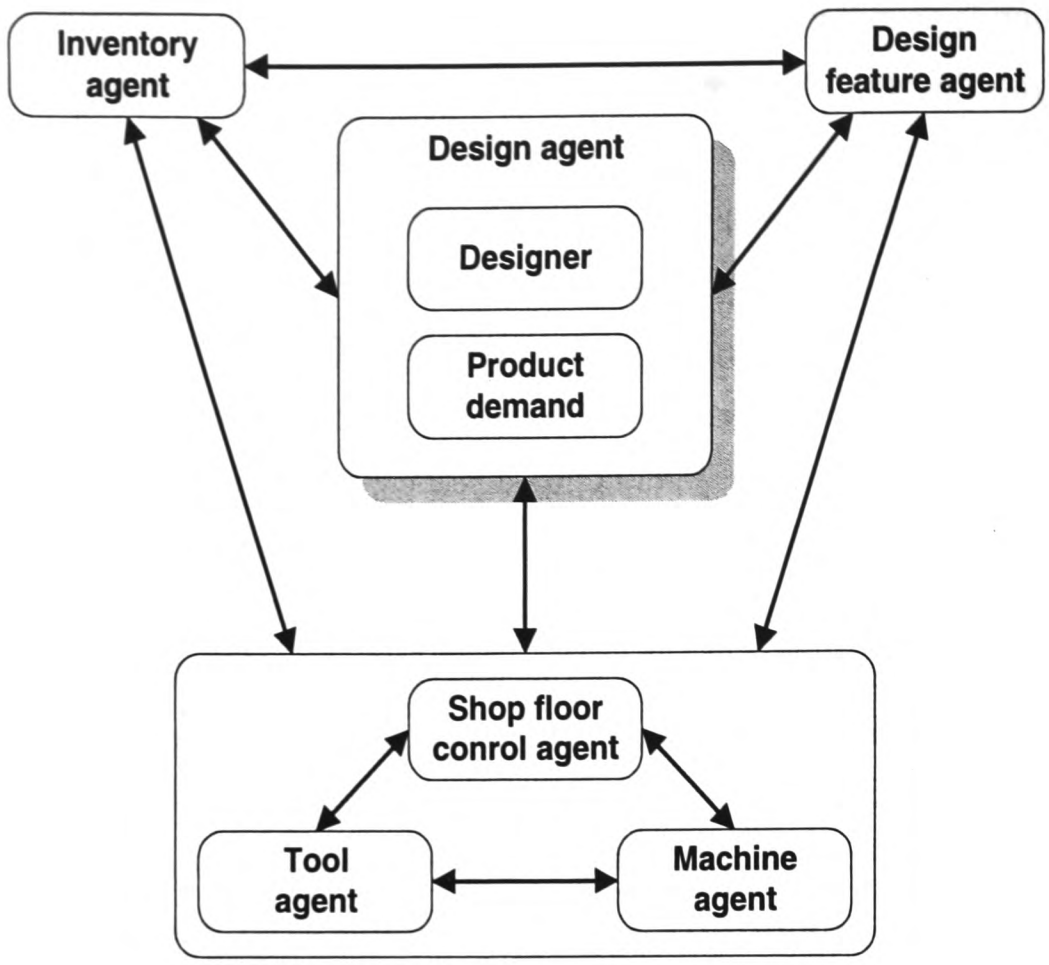


Figure 4-20: ABCDE agent architecture (from [117])

As part of the **GNOSIS** project **[174]**, Balasubramanian proposes a fine-grained approach to the agentification of a feature-based design system **[175, 176]**. This approach is implemented in **ABCDE** (Agent-Based Concurrent Design Environment) and aims to integrate manufacturing design and shop floor control. It proposes the association of each design feature with an autonomous agent. These feature agents initiate dialogue with other agents representing machine tools or stock managers (see Figure 4-20) in order to achieve greater concurrency in the product creation process. Indeed this approach allows design features to interact with shop floor agents as soon as they are created in the CAD system.

For example, a block feature agent can check cost and availability with the shop floor stock manager agent only seconds after being added into the CAD product model. Feature agents inside the system perform validation checks to ensure their manufacturability and notify the designer in case of problems. Subject to successful completion of local manufacturability evaluations, features request the shop floor manager agents to validate global concerns. Machining requirements of features are auctioned by the shop floor manager (see contract-net in section 4.4.1). Successful bids (which include cost/time estimation) from machine tool agents and tool agents ensure manufacturability of the

designed features. When no suitable shop floor agents can be found, the feature is not validated and the design agent is notified. Shop floor agents also maintain their own schedule and can therefore forecast timelines for production of part during early design stages. The novel approach proposed by Balasubramanian harnesses the power of agency to provide automatic design evaluation and true concurrency with the shop floor. However, feature agents do not have control over their geometry and their autonomy is limited to self-evaluation and notification of errors.

4.8 Conclusion

Agent technology covers vast domains of very active research that deserves entire books **[96, 98, 177]** rather than a single chapter. Without aiming for completeness, this chapter has presented an overview of the various issues connected with agency.

The difficulty of formally defining agency was discussed and a more practical property checklist was proposed that characterise autonomous agents as entities possessing important qualities. Namely, delegation of tasks, peer to peer communication, autonomy of actions, pro-action, ability to sense and act upon the environment, and possibly possessing internal symbolic representation of their world.

Important aspects of agency were debated in the form of specific classifications. These taxonomies aimed at shedding different lights on agency in order to give the reader an overview of the field. Particular attention was paid to the paradigms used in decision making. A clear distinction was made between deliberative and non-deliberative agency. Reactive, motivated and planning agents were discussed to illustrate variation of complexity in the decision process. Significant domains of applications were reviewed and different internal agent architecture presented.

The importance of operating inside a community was discussed. This social aspect led to the introduction of agent communication languages, which support most agent interactions. The concept of emergence through agent interaction is a critical property of agent-based systems. It allows designers to model from a local perspective using bottom-up approach.

A survey of significant multiagent frameworks and applications developed to this day concluded this presentation of agent technology.

Chapter 5

Design Features as Autonomous Agents

5.1 Introduction

This chapter presents the feature-based agent-driven approach to engineering design that is the main thrust of this thesis. The main section describes and justifies the agentification at the feature level of feature-based design systems. The concept of an active data model consisting of multiple feature-agents, and assisted by service-agents, is introduced. The important changes to the CAD/CAM system architecture required (and created) by this novel paradigm are discussed. The potential advantages and drawbacks of this approach are critically considered in the last two sections before conclusions can be drawn.

5.2 Fine-grained agentification

The approach proposed in this thesis is for fine-grained agentification of a feature-based design to achieve higher manufacturability of designed parts. The agentification process applied at the feature level is explained and justified in the following sub-sections.

5.2.1 Why use the feature-level?

The use of agent technology in design and manufacturing is a relatively new approach but has already been partially investigated **[179]**. A fundamental difference between the different research undertaken in the area is that of the granularity at which the agents are introduced to the system. Most of the current projects, take a pragmatic view and use agent technology to

add a top layer to existing system **[124, 125]**. This agent layer can take the form of interface agents that assist the user in their interaction with the system. Agents are also being used to glue together the different software packages throughout the design and manufacture process.

A coarse agentification of the CAD system, either at a component level or application level, can bring very valuable new possibilities. In particular it can helps achieve better integration of the various systems used during product design. But it does not directly help to ensure manufacturability of the designed parts. On the other hand, agentification at a lower level, such as primary geometric CAD entities (points, curves, facets), does not seem beneficial as these entities hold little more intrinsic knowledge than their internal geometry.

The approach of this thesis is concentrated on the design feature level. High-level design features such as holes, slots or pockets, hold much of the useful intrinsic knowledge concerning the quality of a design. Indeed it is in the interaction between these design features that the manufacturability of a part lies. It has been seen (see section 2.2.2) that features used in design-by-features system can easily be mapped with template machining cycles (or micro-cycles). It is therefore evident that each of these features contains more than simple geometric data. Indeed, they intrinsically represent a chunk of knowledge concerning the manufacturing process used to produce them. As yet, this potential embedded manufacturing knowledge of design features remains largely unused by current CAD tools.

A fine grained agentification, where each high level design feature is embodied by an autonomous agent can bring the self contained, yet unexpressed, knowledge to the surface in an efficient manner. This level of granularity, which focuses on a feature level, can be justified by results of the work carried out in feature-based manufacturing and design **[15, 56, 61, 178]**. Indeed, these results show that most problems encountered during design, process planning or manufacturing, are consequences of the way features relate with each other **[35]**. The feature-level is considered to be the lower limit of useful agentification as it is the smallest geometric element that still encapsulate rich semantic information related to the manufacturing processes. Moreover, trying to agentify at a finer granularity (edge, point, faces) would involve a tremendous increase in the number of agents needed and thereby in the processing resources needed to support them.

5.2.2 What does a feature agent do?

A detailed analysis of feature agents, their architecture and operation, is given in Chapter 6. However, an overview of feature agent's nature and activity (illustrated in Figure 5-1) is necessary to help understand the concept of feature agent.

Autonomous feature agents are used to embody machining features inside models of mechanical components created inside a CAD/CAM package. They replace traditional static representations and provide high-level services to the designer. They are used at run-time by the system to populate a living agent community that represents the product model being designed. The first part of their autonomous activity consists in analysing their local environment and assessing their fitness against predefined manufacturability rules. These rules are designed to capture known limitations of the machining process (traditional milling and drilling) used to physically create the features. Problems such as tool access and thin sections are among the limitations taken into account. The second part of a feature agents' activity involves attempting to automatically solve the potential problems detected. The solving capabilities of features are held inside a local database of pre-defined strategies that can be selected according to the current environmental conditions. These strategies typically generate geometric transformations to the feature's geometry or its surroundings that resolve a specific problem. Detection of manufacturability problems in its local environment makes a feature agent apply one of its embedded strategies in order to reach a more manufacturable state.

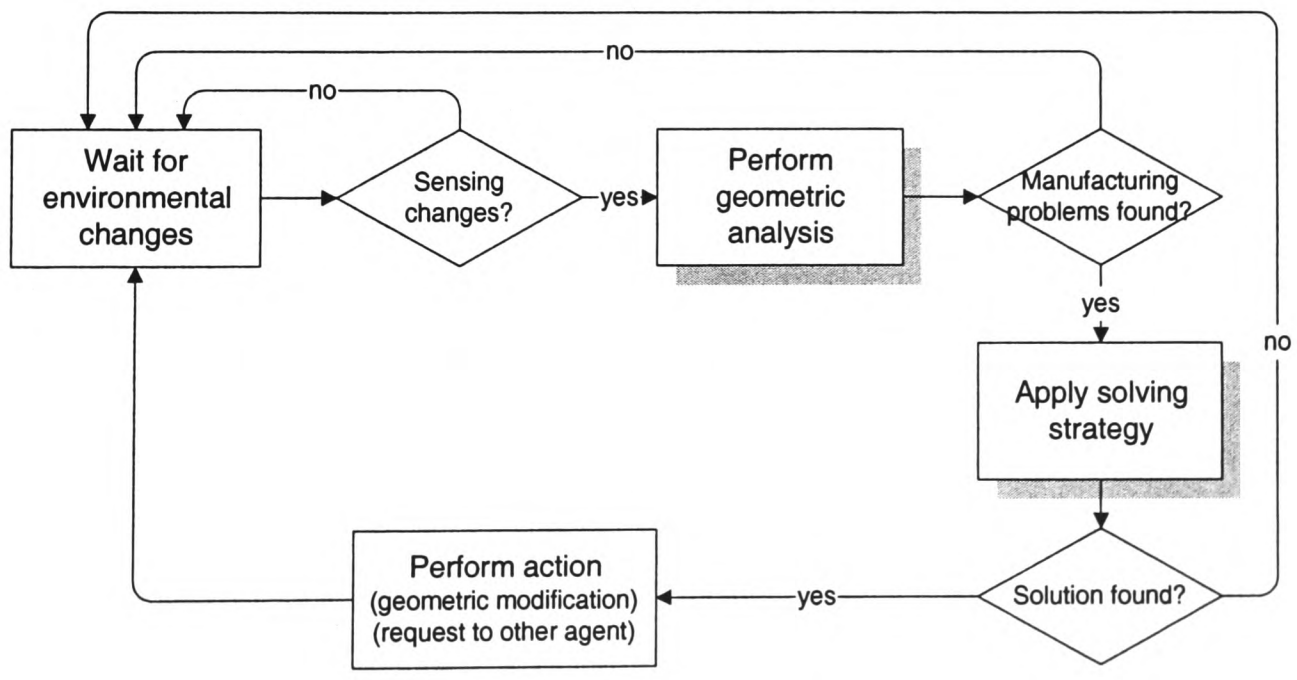


Figure 5-1: Feature agent activity

Figure 5-1 illustrates the general activity of a feature agent inside the system. The agent lies in a dormant state until a geometric change is detected in the model. The agent incorporates the modification inside an internal representation of the model, which is used to carry out geometric analysis and determine the feature manufacturability. If potential problems are detected the feature agent applies a suitable solving strategy in an attempt to reach a new manufacturable state. Whether or not a solution is found to the detected problems, the agent eventually returns to its dormant state until the next perceived model modification. Note that, the behaviour when no solution is found is discussed later in this chapter (see section 5.2.3.b). Feature agents are at the origin of all activity within the system. They autonomously analyse their current condition and apply their embedded manufacturing knowledge to detect and solve potential manufacturability problems in the model.

5.2.3 Delegation to service agents

The concept of feature agents represents the main focus of this thesis. However, the agent-based architecture also introduces service agents that operate in symbiosis with features within the product model.

5.2.3.a Necessity for lightweight feature agents

The introduction of feature agents involves a significant overhead in terms of computing resources necessary to handle a given model. Indeed, feature agents run as individual processes inside the model. As such, they hold not only geometric data but also the processing routines required for their operation. Each feature added to a model requires the creation of an agent object (agents are implemented using OO programming techniques) which includes local data, communication, analysis, and solving routines. Using agents thereby creates a high level of redundancy, which increases system overheads. While this is not a problem for simple components, it can become problematic as the number of features inside a model grows.

In order to minimise these requirements, it is necessary to use lightweight feature-agents that only perform the most common tasks inside the model and rely on external entities to carry out more complex tasks. These external entities are service agents. They offer high-level services to other agents in the system through inter-agent task delegation. A service agent decreases system requirements significantly because it only needs to be instantiated once for the entire system instead of once for every feature. However, communication overheads are introduced by this delegation mechanism, which prevents all activities being

delegated to service agents. Autonomy of feature agents should also be preserved as much as possible which further limits serviced activity.

5.2.3.b Service Agent examples

It has been established that service agents provide expert knowledge to other agents. This knowledge typically complements that embedded in features or offers a functionality useful, but not vital, to the global multiagent community. Three types of services can be offered within the agent community.

- Additional geometric knowledge:

  The limitations imposed on the size of each feature agent (see section 5.2.3.a) make it impossible to embed extensive geometric knowledge within them. The solving capabilities of the individual features, in particular, are limited to basic strategies addressing specific aspects of manufacturability individually. More complete (and complex) geometrical reasoning capabilities can be offered as an optional service to feature agents. This complementary geometric knowledge can be called upon by features in situation where the limited embedded knowledge proves insufficient to ensure manufacturability.

- Functions improving global operation:

  The active agent community representing the product model is a complex system in its own right. It requires a number of mechanisms to support its activity such as inter-agent communication and activity scheduling. All these mechanisms are provided by the system and are sufficient to support the agent community. However, these functions are typically not optimised for a specific application. Specialised service agents can offer improvements and optimisations in these areas.

- New functionality:

  Service agents can be added to the system that adds completely new functionalities. They are few limits to the types of things that can be done through service agents. In the context of a feature-based CAD application, here are a few examples that might spring to an engineering mind.

  ① a display agent that offers visual feedback about the designed component,
  ② a magazine manager that verifies adequate tools exist to manufacture all features,
  ③ a monitoring agent that watches the global health of the system.

Few limitations are strictly imposed on service agent other than being able to communicate meaningfully with other agents. To do this, they must obviously support the communication language common to all agents in the system and use an agreed delegation protocol shared by features.

5.3 Changes in System Architecture

The switch to the agent paradigm brings major changes in the global architecture of CAD systems. The following sections discuss the most important ones.

5.3.1 From passive data to active model

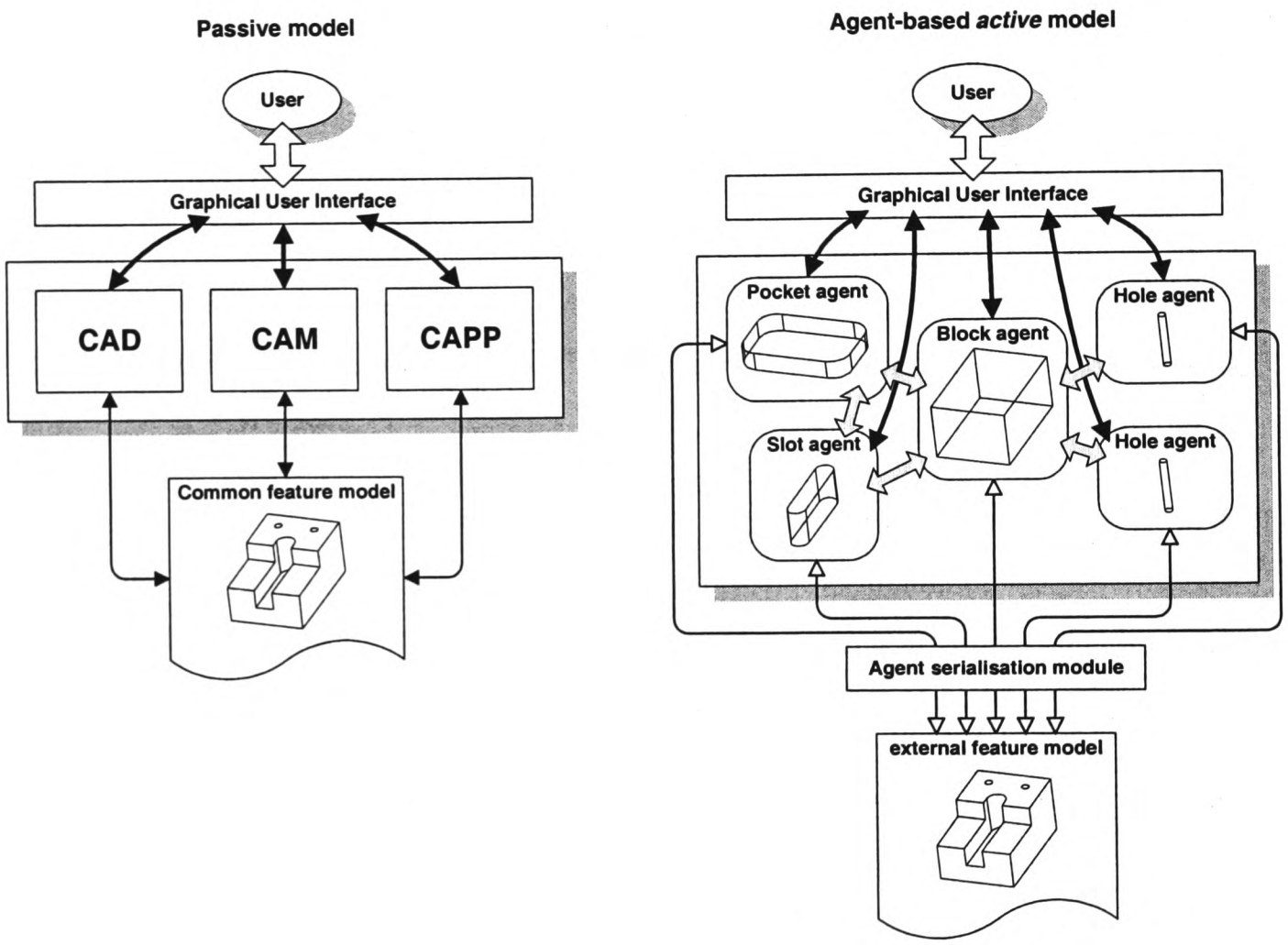


Figure 5-2: Passive and active models

The agentification of features inside a CAD system brings fundamental changes to the global architecture of the system. In particular, the data model used to represent the product is radically transformed as shown on Figure 5-2.

In conventional CAD systems, the product model is represented by a passive data structure separate from the executable applications. Various software modules can access the passive model, interpret the data it contains, apply their processing routines on it and modify it. All

the processing power is contained inside the different modules composing the system. Moreover the system's activity only occurs following specific user commands, following the client/server model of user interaction. The user does not interact directly with the model; instead they interact through the available modules which in turn modify the common feature model.

Using autonomous agents to embody design features radically transforms the way the design system is structured. A community of autonomous feature agents replaces the traditional passive data structure used to embody the product model. This community represents an *active product model* that works on behalf of the designer. The activity that used to be forced upon the product model by external modules is now self contained in the agent community representing a design. Indeed, features are not statically stored as a few bytes of passive data; instead each feature inside a mechanical component becomes an autonomous agent capable of setting its own agenda and taking initiative concerning its geometry. The use of agents to embody features puts the global activity of the system inside the model itself. In turn this allows the user to interact directly (via the necessary GUI) with features making the model.

Although the active model is radically different in nature from the passive data, the multiagent system still allows saving and loading of external passive data into, and from, external file. This is achieved through an agent serialisation module. While saving component models, this module can dynamically interrogate individual features about their geometric properties and save the results using traditional passive data structures. Loading of conventional feature models is more complex as it involves the dynamic creation of feature agents and their insertion inside the running agent community.

5.3.2 Parallel processing

The parallelisation of processing tasks is a very active field of research **[179]**, which proposes to decompose a computing problem into loosely dependant elements. These elements can be processed separately, and in parallel on several processing units. This approach maximises the throughput of computer systems, thereby minimising time taken to find a solution. Parallel processing is seductive because, it allows one to take advantage of the theoretically boundless power (simply add more processing units) of parallel computer systems.

The agentification at feature level described in this thesis provides a natural decomposition of the CAD system that is well suited for parallel processing. Each autonomous agent is only loosely dependent with others (through peer-to-peer communication) and is able to run within a parallel environment. Indeed, autonomous agents inside the system allow geometric processing to be carried out locally by individual agents without global control. Moreover, each autonomous agent can be implemented as an independent process inside a computer system.

One could consider potential parallelisation for the two architectures illustrated in Figure 5-2. The traditional approach could benefit from having each module (CAD, CAM and CAPP) running in parallel. However such a coarse-grained decomposition only applies to the processing software modules and the system does not scale with model size. Compared to the traditional passive product data, the multiagent active model is thereby a much better candidate for parallelisation. Its particular strength lies in its built-in scalability that allows the system to take advantage of highly parallel computers. Because each feature agent can run in parallel processes, they can each run on a different processing unit. Therefore, increasing the number of features in a model does not significantly degrade system performance as long as enough processing units are available.

Parallel computing is a complex domain, which brings it own difficulties, Interprocess communication, in particular, becomes problematic in highly parallel computers and limits the actual number of processing units that can be efficiently clustered together. Yet, the advent of affordable parallel technologies (Symmetric MultiProcessing **[181]**, Beowulf clustering **[182]**) makes parallel computing a seductive solution for engineering purposes. The feature-based agent-driven approach described in this thesis offers a natural and elegant parallelisation route for mechanical design systems.

5.3.3 Distributed processing

A distributed software system describes a group of programs offering integrated functionality despite running on separate computers, and possibly in distant geographic locations. The computer systems involved in supporting distributed applications are linked together by an information network.

In the context of concurrent engineering, distribution has been investigated at application level as a means of achieving real concurrency between the different phases in the creation of new products. Distributed computing offers the possibility of having multiple engineers

working on a product at the same time in different locations. It has been actively pursued by manufacturing industry, which strives to achieve truly concurrent engineering. Agent technology has been successfully investigated as a means of integrating distributed engineering system **[124, 173, 183, 184, 185, 186]**. However, this approach is still based on the traditional modular approach shown in Figure 5-1.

The parallelism provided by autonomous agents has been discussed in 5.3.2 and naturally leads to the issue of distribution. The potential distribution of agents across different working posts or even sites is a great asset of the feature as agent approach. It offers unmatched flexible distribution mechanisms. The only prerequisite for the creation of distributed multiagent system is the existence of an inter-agent communication layer that supports physical distribution of agents. Such communication languages already exist in the agent research community (see KQML in section 4.5) which makes distribution possible.

An additional property of certain agents further increases the potential for distributed multiagent systems. Agent mobility is concerned with the various technological mechanisms that allow a software agent to transport itself from one host computer to another **[187]**. It is obvious that this concept of mobile agency further increases the flexibility of MAS in the context of distributed computing by allowing run-time re-organisation of agents.

5.3.4 Time continuity

A major difference between most conventional and agent-based applications is the influence of running time over proper operation. Conventional applications are mostly, *execute*-and-*terminate* programs. They are launched by users, perform their task and terminate. If needed again in the future they are simply launched again. Time has little meaning for such applications. When launched, they evaluate the current situation, apply their operating knowledge and terminate. There are some exceptions to this, background processes (often named daemons) are conventional programs that run continuously and perform tasks (networking, database) in response to low-level events. However, such daemons perform their task equally well immediately after being launched as after a month of uninterrupted operation. In that sense, time continuity does not affect their function.

Agents are dynamic entities that adapt to the environment they live in. They build an image of their environment through their sensing capabilities (mostly inter-agent communication in the case of software agents) which is then used to dynamically generate their behaviour. This internal representation of the world is not something that agents are

born with. Instead it is built over time by the agent and reflects accumulated knowledge about the environment. Because of this, the performance of agents is affected by time. A newly created agent possesses no knowledge of its environment and needs to build up its internal image of it (learn its environment) before generating its behavioural responses. So, time continuity is important for agents. In identical situations, a newly created agent might be unable to properly operate, while an *older* one would behave without difficulties. Note that time continuity does not concern purely reactive agents, that usually do not possess an internal representation of their world. It does affect higher forms of agency, which rely on more cognitive processes, and thus on accumulated environment data, for behaviour generation.

Because of their reliance on an internal representation created with data collected over time, agent-based applications are *execute*-until-*terminated* programs. Such applications are started once and run uninterrupted for as long as it is practical to do so. While running, agent-based systems provide an agent-friendly framework in which to work. They are populated with autonomous agents that remain active for long period of time: until specifically deleted or the system is shut down. During the duration of their *lives* inside the MAS, agents build and maintain their individual view of the world they live in. They use this representation to assess their own situation and take actions if necessary. Thus, an *old* agent is usually more efficient that a *younger* one because it has better knowledge of its current situation.

Time continuity is an important factor in an agent's capability to perform their task efficiently. Unlike most conventional software, agent-based applications can not *execute*-and-*terminate* on user demand. To behave correctly, agents require local knowledge of their environment that can only be learned by living inside a MAS for a period of time.

5.4 Potential benefits of feature-agents

Using autonomous agents to implement design features brings a number of benefits. Some of these are the usual expected benefits of agent-based systems such as scalability, extensibility, fault tolerance and modularity (see Chapter 4). Other added qualities are more specific to the application of the agent paradigm to mechanical feature-based design.

5.4.1 Continuous design evaluation

It has been explained that, using feature agents, the product model becomes the centre of the system activity (see section 5.3.1). Each feature inside the agent community is an autonomous entity capable of planning its own actions according to its perception of the situation. This autonomy given to features allows the system to perform continuous analysis of the design.

When a geometric change occurs (creation, deletion or modification of features), it is automatically perceived across the model by all neighbouring feature agents. Each feature maintains an internal representation of its environment in which the perceived changes are incorporated. This internal representation is a snapshot of the feature's surrounding and is used to analyse the local manufacturability of the feature. The autonomous local manufacturability analysis is triggered by the perception of changes in the system and need not be invoked by the designer, or any other global control structure in the system.

The consequence of the autonomous analysis of local conditions performed by each feature agent, is the creation of a model that gives spontaneous feedback concerning the impact of changes in the model. The community of agents representing the model responds to every geometric modification with an immediate evaluation of the manufacturability of the model.

5.4.2 Self-correcting model

Autonomous agents are a powerful paradigm because they allow the introduction of initiative. In the case of feature agents inside the model of a mechanical component, this initiative is used to permit features to perform certain tasks on behalf of the user. In particular, features can try to resolve various types of manufacturability problems, leaving the designer free to focus on more important design issues. A certain degree of autonomy can be granted to each feature agent inside the model they inhabit. Geometric features are given full or partial control over their own geometry, they can modify themselves or request modification of others. This allows simple manufacturability problems in the design to be autonomously solved by the agent community, granted the solution doesn't violate constraints given by the designer.

The autonomous geometric analysis performed by a feature (see section 5.4.1) yields a manufacturability status that reflects the current situation of the feature relative to others in

the model. When full manufacturability is not achieved, features attempt to find a local solution by applying embedded solving strategies. In most situations, geometric transformations can be found that eliminates detected manufacturability problems. The proposed geometric modifications must be checked prior to their application. If they do not violate any constraints previously expressed by the designer, the feature performs them on itself or sends a request to others. This control given to features over their geometry creates a *living* model that performs autonomous modifications to the design to ensure the components' manufacturability.

In most cases, no user intervention is necessary to resolve the detected manufacturability problems. The self-correcting model can be trusted to handle a family of well-understood manufacturability problems, thereby removing the burden of manually dealing with them. The designer can focus on more demanding aspects of the design, leaving the model to deal with less critical consequences of any geometric changes introduced.

5.4.3 Localised, problem-focused activity

Not only does each agent achieve continuous analysis of its individual manufacturability and perform autonomous geometric transformation, it does it locally rather than globally. The new agent-based architecture of the system only focuses on critical areas of the design. It naturally avoids wasting resources testing unmodified parts of the model. Indeed, agents only respond to modifications in their local environment. Most of the available activity in the product model is therefore directed at detecting and solving potential manufacturability problems rather than re-assessing satisfactory portions of the design.

Because it uses the agent-based approach, the design system is naturally incremental as all analysis and computations are performed locally at feature level. The autonomous activity of a feature is triggered by geometric changes in its immediate surrounding. A modification made on a feature might generate autonomous responses from neighbouring features but could be ignored by other features. This focusing of agent activity inside the model is based on space occupancy and geometric interactions. Each feature intrinsically possesses a space of influence that marks the limits of its local environment. Any feature partially or fully occupying, this space can potentially affect its manufacturability. Others are simply discarded because they are not local neighbours. This principle of locality allows the agent community to focus its activity onto the areas of a design that are being actively modified.

Moreover, the solution knowledge of features is also applied locally, within each feature and uses their internal representation of the component. This helps focus the system further by containing the solution activity inside features that detect potential manufacturability problems.

5.4.4 Increased interactivity

Most conventional CAD systems adopt a client/server scheme for user interaction. The user (client) must request the execution of a command to the application (server), which complies and communicates the results back to the user. The efficiency of this scheme is not challenged but it creates an interface between user and machine that lack interactivity.

The agent paradigm is based on delegation, autonomy and communication. These qualities allow for more interactive systems. Indeed, agents are able to initiate dialogue with other agents when required. On a peer-to-peer communication scheme, autonomous agents in the model are able, for example, to point out potential problems to the user even in the absence of a specific request being issued. Also, if a feature agent is unable to fully ensure its manufacturability by using its limited embedded knowledge, it can initiate dialogue with the user, or another agent, and delegate all or part of this task.

The ability of agent to take the initiative and communicate on a peer-to-peer basis offers greater interactivity than conventional client/server approach. In particular, it is possible to consider the human designer a special type of *service agent* within the system. The user is in this way integrated inside the agent community. It can therefore engage in peer-to-peer communication with other agents and benefit fully from the autonomous activity of features.

5.4.5 CAPP pre-processing

Within the multiagent community representing the model, each feature performs many geometric tests during the course of the design and stores the results inside its local representation of the environment. Features use their internal representation of the component to assess their manufacturability and apply their solving knowledge. This accumulated local knowledge is present in all active models and could be used to ease the process-planning task.

Each feature is able to provide much more information about the model than pure geometry. Indeed, by volunteering the partial results contained in its internal database a

feature agent is able to provide useful data concerning local manufacturability issues such as tool access. Local beliefs concerning indirect tool access (through one or more other features) can be easily expressed as precedence constraints in the machining sequence. The process planner usually determines these constraints by performing geometric reasoning on a purely geometric model. If used in conjunction with the proposed multiagent feature-based design system, the process planner could use this pre-processed information. The process planning system can therefore focus on other planning problems.

5.5 Drawbacks of feature-agents

The use of autonomous agents has drawbacks. The most significant difficulties introduced by the feature as agent approach are described in the following sections.

5.5.1 Communication load

The agentification of features proposed in this thesis creates a non-hierarchic community of autonomous entities. The potential for exponential increase of communication load was discussed in 4.6.2. Yet the feature as agent approach provides a convenient way to control the amount of peer-to-peer exchanges taking place within a model. The principle of geometrical locality applies to feature agents and can be used as a basis to limit communication load. Features can be clustered according to their geometric locality by using techniques based on bounding boxes or space partitioning (see section 6.3.4.d). The dynamically created geometric clusters help keeping the communication load of the system under control.

5.5.2 Sub-Optimality of solutions

All the solving activity carried out within a component is performed locally by agents. It is this local nature of agent's operation that creates robust and flexible systems. Feature agents are localised based on their geometry and need not have knowledge about the complete design to perform useful actions on behalf of the user. However this same locality of analysis and action prevent the system from seeking optimal solutions to detected problems. Indeed, the solving knowledge is applied using the local knowledge of features without consideration for the global state of the component. Therefore, there is no way for the system to assess the optimality of local solutions. Instead agents attempt to ensure that the component remains in a satisfactory manufacturable state, not the optimum manufacturable state (see section 4.4.1).

Because feature agents were never intended for manufacturability optimisation, this is not seen as a critical weakness of the system.

5.5.3 Model Instability

The agentification of features has created the active model described in 5.3.1. The active model is given a degree of control of its geometry through autonomous feature modifications. This autonomy, almost self-determination, is a core contribution of features as agents. However, it is in potential conflict with the model stability required by engineering design.

In a traditional CAD environment, the designer is in full control of the designed component. Only he can modify the geometry of the features within a model. If a change is made that compromises the component function or manufacturability, it is the designer's responsibility to either cancel the modification or deal with the consequences with further modifications. This approach provides a stable (passive) model but requires the designer to manually address all aspects of the design.

The agent-driven approach proposed in this thesis, creates a potentially unstable active model that works autonomously on its own geometry. The manufacturability aspects of the designed component are handled automatically by the feature agents instead of manually by the designer. The feature agents perform geometric modifications on behalf of the user to ensure manufacturability. Such autonomous modifications of feature, ensuring manufacturability, are the main benefit of feature agents. However, it is possible that such autonomous changes go against the designer's intention. It is also possible that a self-reliant modification induce multiple others in a chain reaction that compromises the overall quality of the design.

On one hand, the autonomy given to features helps the designer by removing the burden of manually dealing with the consequences (from a manufacturing point of view) of geometric modifications. On the other hand, this very autonomy potentially hinders the designer's efforts by cancelling his changes or destroying satisfactory areas of the design. Clearly, a compromise has to be struck between autonomy and stability, which allows features to work on behalf of the designer without going against its interests. This compromise can be achieved through various mechanisms controlling feature's activity. It should be possible to apply loose constraint to features that set limits to their autonomous transformations. It

should also be possible to disable a feature's solving activity for satisfactory areas of the design.

5.6 Conclusion

This chapter has presented the concept of feature agents that is central to the work described in this thesis. By using autonomous software agents to embody individual design features it is possible to radically transform the way CAD/CAM software is built and operated.

The traditional data structures used to store product model is replaced by a community of feature agent that is the centre of activity of the system. A novel active model is thereby created that brings new functionality to feature-based design. Each feature agent is made responsible for its geometry and pursues individual goals of ensuring its local manufacturability. The adoption of autonomous agents also creates great potential for parallel and distributed execution of the active model.

Real-time analysis and self-correction of the model are the most significant advances brought by the adoption of the agent paradigm at feature level. A mechanical component designed using feature agents, is able to assess and ensure its manufacturability in real-time, without intervention of the designer. Feature agents are given a degree of control over their geometry and are trusted to solve a number of well-understood manufacturability problems. A self-correcting model is thereby created which performs geometric modification (related to manufacturing considerations) on behalf of the user. The active model deals with the non-critical consequences of changes made to the model, leaving the designer free to concentrate on real design issues.

Chapter 6

Implementation: MultiAgent Design System for Manufacturability (MADSfm)

6.1 Introduction

This chapter describes the implementation of multiagent design systems, and test-beds used to investigate the features as agents model proposed. First, a number of preliminary tests are described, which elicited much information on different aspects of multiagent architectures and what they can bring to CAD/CAM systems. Then, the final implementation of a MultiAgent Design System for manufacturability (MADSfm) is described in details.

6.2 Preliminary tests

This section presents part of the preliminary implementation work carried out during the course of this research. Although not directly part of the presented contribution, the work is instrumental in getting a fuller understanding of various important issues related to it. In particular, it provides valuable insight into problems specific to agent-driven application.

6.2.1 Parallel architecture in C++

When this implementation work commenced, no multiagent development environment was readily available to satisfy the specific requirements of a computationally efficient (in

terms of speed and memory usage) multiagent system. Development was therefore undertaken of a set of C++ classes implementing a multithreaded[5] agent architecture.

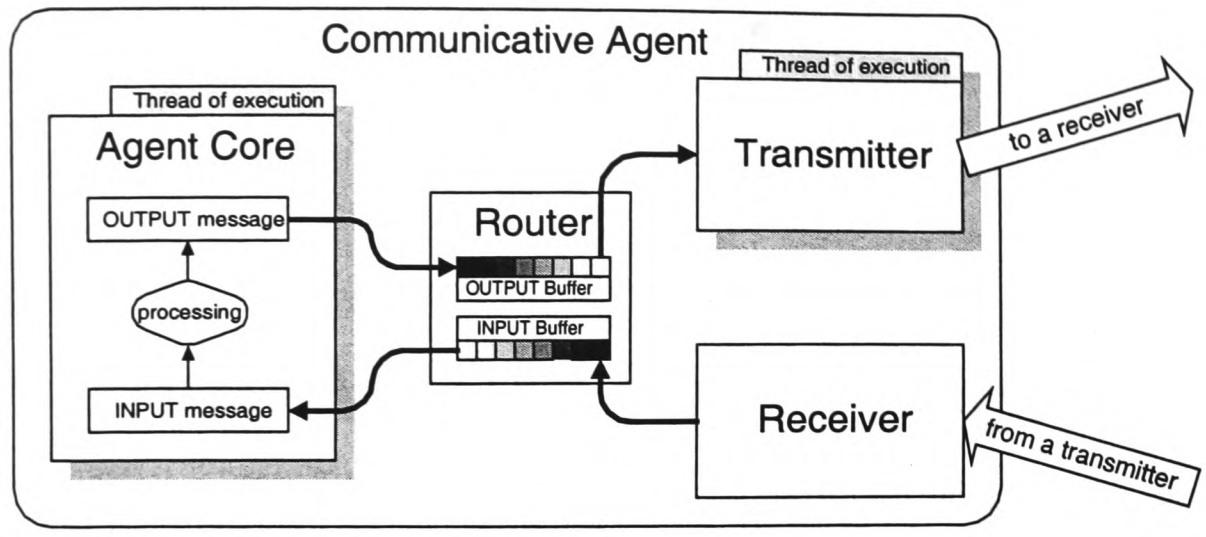


Figure 6-1: C++ agent architecture

The initial objective was to develop an infrastructure that offered support for truly concurrent, communicative and asynchronous agents. The resulting classes were used to created a Windows NT® application that supported dynamic creation of multiple agents. Each agent is composed of several threads of execution running inside the main application memory space. Figure 6-1 illustrates the architecture of the agents created using the C++ classes.

Several important lessons were learnt from implementing an in-house agent-based *engine*. Inter-agent messaging is supported through an implementation of a subset of KQML (see section 4.5) agent communication language. Because KQML is designed to be an easily generated and parsed format, the processing overhead is small and KQML proves to be well adapted for lightweight agents. Peer-to-peer communication is supported between agents but facilitator agents are also used to provide high-level services such as name/address resolution, multicast, and broadcast. Asynchronous operation of software agents revealed the necessity to separate communication from other agent activities. Indeed an agent's core should not need to interrupt its activity whenever transmissions are being made. A communication layer was thereby created as a plug-in component (called a router) for agents. Because agents deal only with router and never directly with the physical communication medium, communication delays and errors do not affect their internal activity. Finally, the challenge of creating truly concurrent software agents exposed some limitation of current computing techniques. In particular, the difficulty to handle parallel access to shared resources proved a great burden. Autonomous entities living in common environment (which

[5] Threads are the smallest execution entities recognised by the operating system scheduler.

is almost the definition of agency) always end up accessing shared resources concurrently. In the case of physical world, simultaneous actions on a single object is sometimes possible. In the virtual world however, such concurrent access is not possible. The environment needs to be represented symbolically and held in a memory space (RAM or disk) which currently does not support simultaneous operations (attempting it often result in a crash). This hardware limitation forces programmers to ensure exclusive access to shared resources using programming technique such as *mutexes* and *critical sections,* which guarantee exclusivity though the use of virtual locks and keys associated with resources and processes. In multiagent applications, concurrent accesses to common resources can be very frequent and the overheads of such programming techniques become significant. Moreover, the programmer must ensure that deadlock situations, in which one agent requires a resource that is never released by another, do not arise. To put it mildly, in the case of complex multiagent application, exclusive access to shared resources becomes very hard to track.

6.2.1.a Purely reactive Agents

The simplest form of agency is obtained with reactive agents. Reactive agents only require a set of reaction rules and a perception mechanism in order to function. The implementation of reactive agents represents a first step that must be taken in the initial testing of any agent-based engine. A reactive agency test is thereby conducted to demonstrate the soundness of the in-house agent framework. It is based on collision avoidance, which is a common behaviour in reactive agency. Two-dimensional circles are created as agents. They live inside an infinite plane and are behaving so as to ensure they do not intersect with each other. Circle agents only hold their own position and radius as internal variables *(see intr*insic world representation in 4.3.1.b). No extrinsic world representation is provided and agents rely totally on geometrical information being submitted to them through their standard communication channel. They are equipped with a specific set of rules designed to generate the desired behaviour:

```
1. IF (created) THEN (broadcast position and radius)
2. IF (modified) THEN (broadcast position and radius)
3. IF (intersection detected) THEN (move away)
```

Rule 3 represent the main behaviour of agents and is responsible for provoking avoidance reactions on their part. The other rules are communication rules that allow the propagation of changes. Rule 1 guarantees that newly created circle agents make themselves known to the rest of the agent community and thereby allow detection of new intersections. Rule 2 makes

agents volunteer their position and radius after each modification, thereby propagating changes to neighbouring agents.

When agent A receives a message (which always consists of a position and a radius) from agent B, it computes their relative centre to centre distance D_c, then compares it to the sum of their radiuses R_Σ. If $D_c \leq R_\Sigma$, agent A detects an intersection with agent B and applies rule 3. The *move away* behaviour is consequently easily realised by agent A, who modifies its position with a translation along a calculated *escape* vector (the centre to centre vector). The distance of translation is also calculated in order to achieve $D_c = R_\Sigma + \delta$ (δ being a predefined small "clearance" distance).

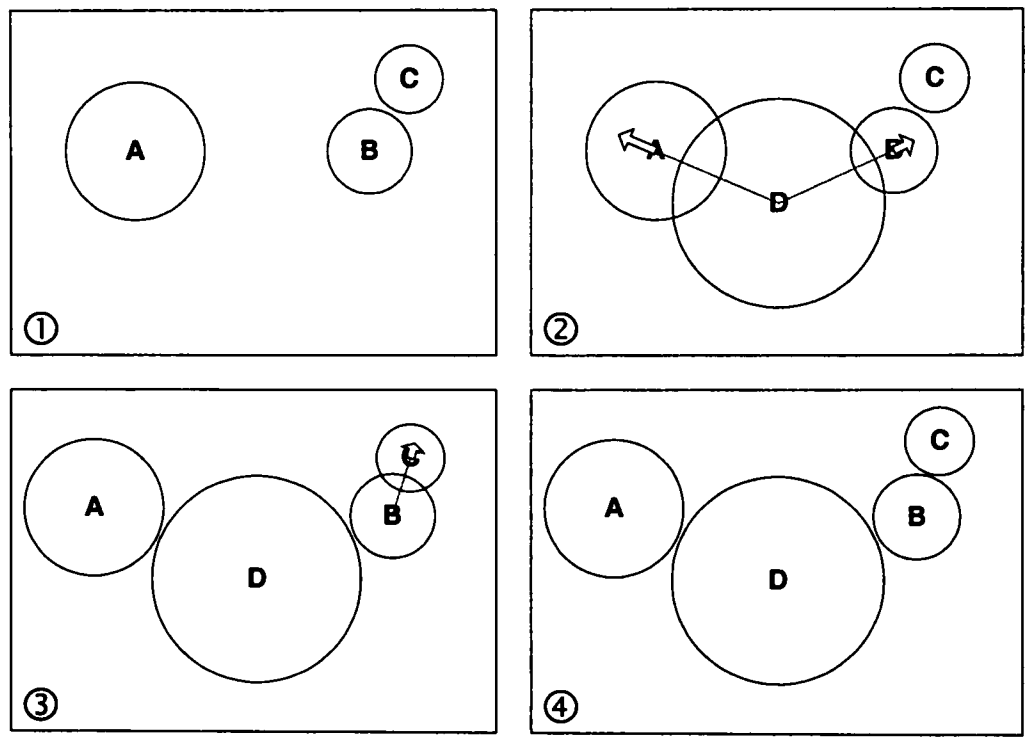


Figure 6-2: reactive agency, example of non-intersecting circles

Circle agents are added to the system one after the other. Each agent applies the same set of rules and a consistent global behaviour is obtained. Figure 6-2 shows the initial stage of a test run. In Figure 6-2①, agent A, B and C are created and no activity exists in the system since there are no intersections. Circle agent D is added in Figure 6-2②, which intersects with existing circles A and B. Agent C apply rule 1 and broadcast it geometric properties. On reception of this message, both A and B detect an intersection, apply rule 3 to change their position to Figure 6-2③, then rule 2 to broadcast their new position. On reception of agent B's broadcast, C applies rules 3 then 2 to eliminate an intersection, which bring the system to state Figure 6-2④. Because no intersections exist in this state, no new activity is triggered by the reception of agent C's final broadcast.

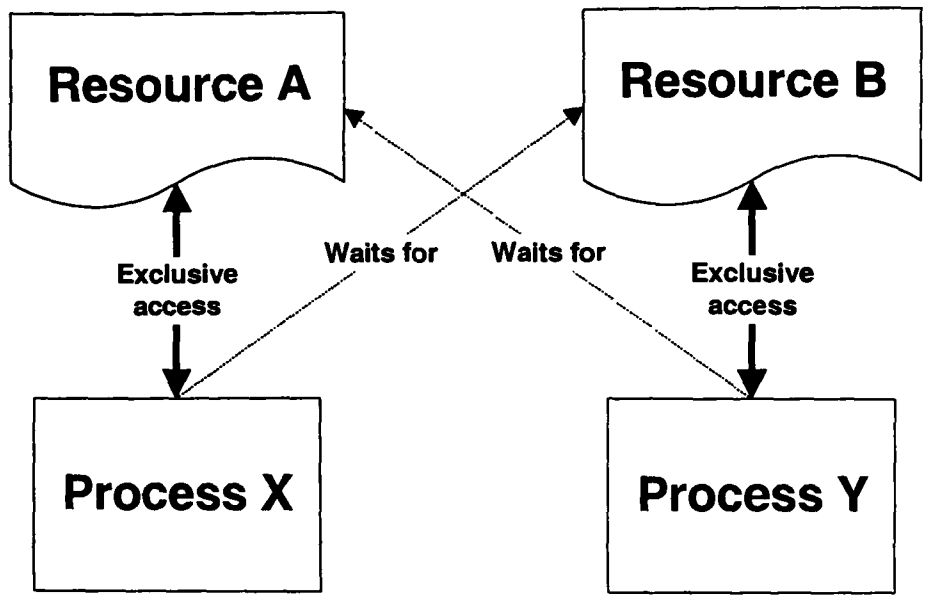


Figure 6-3: Example of deadlock

The system displays a coherent global behaviour. Reactive actions are taken by agents to avoid intersection. Some actions result in new intersections being created. Because agents broadcast their geometry after modification, these new intersections are immediately eliminated and it is possible to witness wave-like propagation of changes within the system. However, very important issues are hi-lighted by this simple test. Firstly, the *communication load* inside the agent community grows exponentially, reaching very high values, as the number of agents in the system grow. This can be partially explained by the fact that only broadcast transmission are used and highlights the importance of efficient communication methods between software agents. The other issue raised during this test is the problem of *livelocks*. While *deadlocks* are recognised obstacles that needs to be avoided in conventional programming, livelocks represent a new type of problem that appears in MAS. Deadlocks describe a situation where two, or more, processes are locked in a mutual waiting state. It usually arises when several processes require access to shared resources but can never obtain it (see Figure 6-3). For example, process X holds exclusive access to resource A and requires access to resource B. At the same time process Y holds exclusive access to resource B and requires access to resource A. The two processes await indefinitely in this crossed dependency and the system is deadlocked.

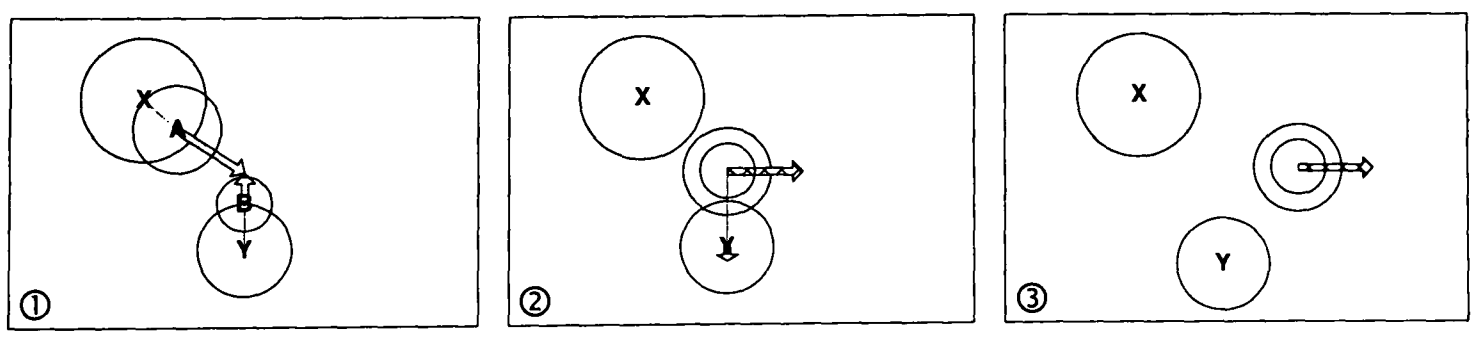


Figure 6-4: Example of livelock with non-intersecting circle agents

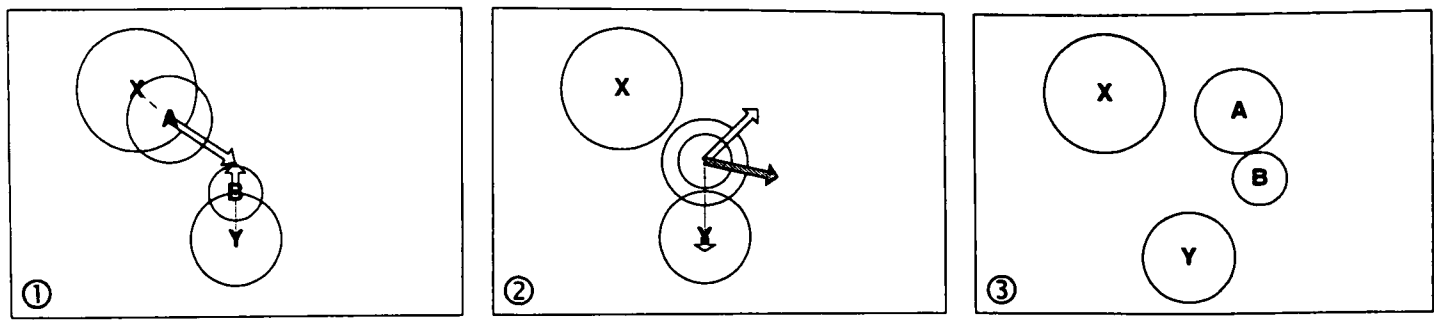


Figure 6-5: Livelock prevented by adding randomness to the *escape* vector

Livelock describe a situation opposite to that of deadlock, in which the activity of several processes is mutually maintained in a never ending event loops. In the case of autonomous agents, the possibility of livelock situations is real because no global control is present to detect and prevent them. Reactive agents are particularly prone to livelocks as the non-intersecting circle test showed. Indeed the technique used to resolve intersections can creates problems when several circles share the same position. In the specific case of concentric circles, the *escape* direction that is usually deduced from the centre to centre vector is undefined. An initial (and naïve) implementation decision is to pick a predefined direction (the X-axis) as the escape vector. This choice however can lead to livelock situations. Indeed because all agents apply the same rules, when faced with identical situations, they take identical decisions. When sharing the same positions, two circles will therefore solve the intersection using identical moves, re-creating an identical problem at a different position. This livelock visually looks like two circles endlessly chasing each other across the infinite plan (see Figure 6-4 ② and ③). Inserting some degree of randomness in the agent's moves can solve this livelock. Instead of picking a pre-defined escape direction for concentric circles, it is possible to choose a pseudo-random direction than prevents the *chasing* behaviour (see Figure 6-5).

The agent C++ classes show their ability to handle reactive agency with no major difficulties. In particular, they prove capable of handling heavy communication load without damaging the system's responsiveness. The manifestation of livelocks inside such a simple MAS confirms the importance of considering special cases with care when designing agent's behaviours. It should also be noted that they are a direct consequence of the true concurrent nature of the C++ agents obtained. If agents were acting sequentially (in a round robin manner) instead of concurrently, livelocks would be less likely to occur.

6.2.1.b BDI style agents

The limitations of reactive agency are rapidly demonstrated by the initial test with non-intersecting agents. In particular, the system stability proves less than adequate for a CAD environment. Because the circle agents possess no mental state other than their own position and radius, they are unable to take "*optimised*" decision. For example, when eliminating one intersection, agents do not take any account of the potential consequences of their actions. It is thereby possible that the removal of one intersection results in the creation of several others. This might not be a problem in some applications but is not acceptable in CAD where partial solutions should be preserved as much as possible. It is also doubtful that purely reactive behaviours could be designed which would permit adding useful functionality to a CAD system.

The second step in the preliminary work was to experiment with "higher" forms of agency. Above all, the addition of mental states and more deliberative behaviours needs experimentation before selecting the type of agents that could be used inside a feature-based CAD system. The BDI architecture discussed previously (see section 4.3.5.f) provides a well-designed decomposition of agent's mental states. The second preliminary implementation is thereby that of motivated agency using an architecture similar to the BDI architecture.

A test system was devised for solving geometric problems similar to some that might occur inside feature-based CAD systems. This test consisted of two types of three-dimensional blocks living inside an infinite space and is inspired by design with machining features. Positive blocks correspond to volumes of material, while negative blocks represent material removals. Each type of block possesses specific behaviours that address simplified problem encountered in feature-based design. Positive blocks are equipped with a single behaviour, which is to ensure that they do not intersect with other positive blocks (as it would be physically impossible). Negative blocks enjoy two distinct behaviours, which endeavour to guarantee their usefulness as machined volumes. Firstly, they ensure that they intersect with at least one positive block (to guarantee the removal of some material). Secondly, they attempt not to intersect with other negative blocks (to avoid machining the same volume more than once).

	Positive block agent	*Negative block agent*
Desires	Collision count = 0	Collision count = 0 Presence count > 0
Beliefs	Collision count	Collision count Presence count
Plans	Solve collision	Solve collision Solve presence

Table 6-1: Mental state of block agents

Table 6-1 shows that both agent types are motivated using beliefs representing the counter for geometric intersections and desires (goals) fixing acceptable values for these counters. Collisions represent geometric intersection with a block of same materiality while presence represent geometric intersection with a block of opposite materiality. Negative blocks use presence counters to ensure removal of material but not positive blocks, which do not actively seek to be machined. The plan database is simplified and represented by routines (procedures) dedicated to reaching individual goals.

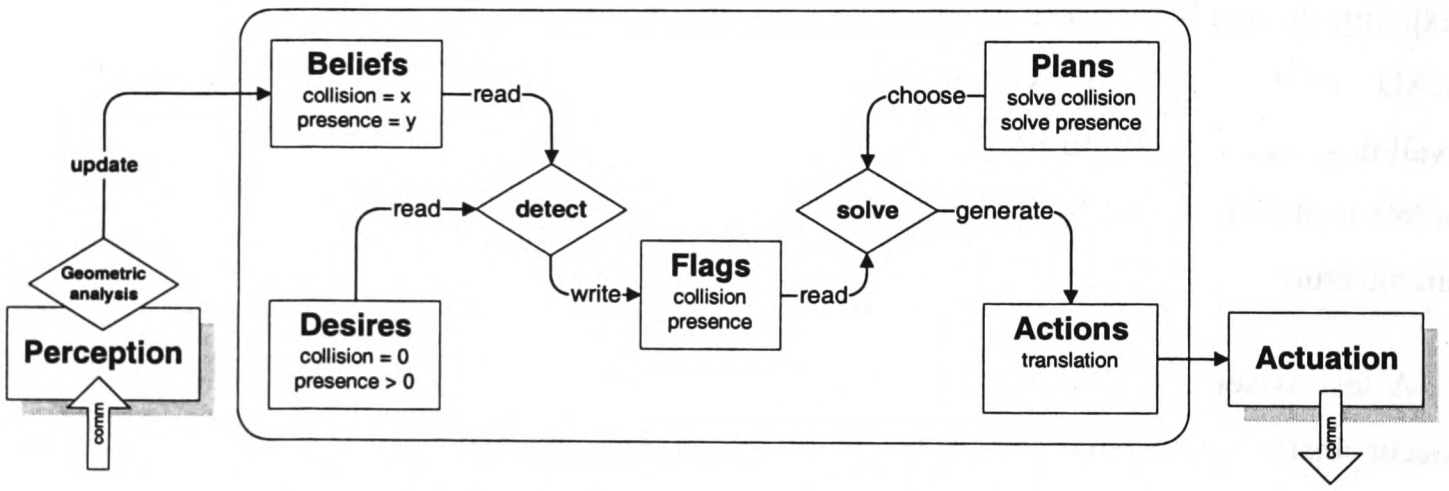


Figure 6-6: Internal operation of negative block agent

Once again, no extrinsic world representation exists and agents rely on other agents to send their geometric properties through communication. After reception of such a message by the perception (communication) module, the incoming geometric data passes through an analysis process, which performs intersection tests. Agents maintain a list of other agents and their associated intersection status. This list is used to calculate collision and presence counts that are stored as internal beliefs. In parallel to this geometric analysis activity, agents actively seek the fulfilment of their desires as illustrated in Figure 6-6. Dynamically updated beliefs are compared with the pre-defined desires to detect mismatches. The results of these comparisons are stored as internal flags marking the need for action. The dedicated solving procedures are applied, when flags are up, which generate geometric transformations (translations) for the agent. After a translation has been performed, modified agents

broadcast their new geometric properties to the rest of the community, thereby allowing propagation of changes inside the system. It is clear that this architecture is not truly of BDI type. In fact it used the BDI scheme in order to trigger reactive modules instead of planning modules. Such a scheme could be defined as *motivated* reaction.

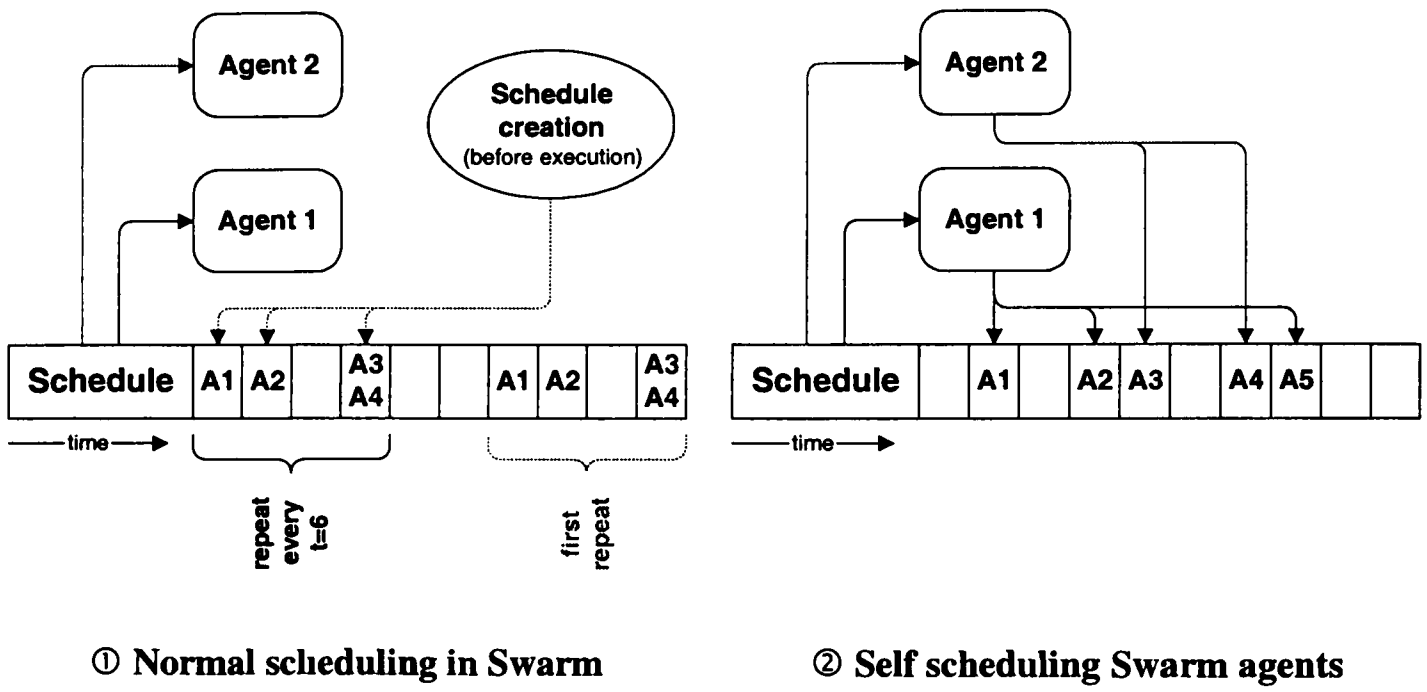
The resulting system allows the creation of blocks with arbitrary size and materiality, which are added one after the other. The agent community displays autonomous activity in solving any collision or presence problems. Positive blocks always detect collision and correctly take actions to eliminate them. Negative blocks also operate as planned; both avoiding collision and seeking presence. They prove that several goals can be pursued in parallel using motivated agency. The major advantage of this motivated approach is the addition of a mental state that increases agents' behavioural stability. Indeed, because agents remember their intersection status with other agents they can act only when necessary. For example, a negative block will not trigger its presence seeking behaviour until it believes its presence counter to be zero. Environmental state rather than events now drive the activity of agents.

An important problem surfaced from implementing these motivated agents. A phenomenon that makes agents over-react to their environment was observed. For example, two positive blocks that intersect tend to move apart from each other more than necessary when solving their conflict. In fact, conflicts are always solved twice (once by each protagonist). This over-reaction results from the combination of the true concurrent nature of agents and their individualism. Indeed, these agents trigger their actions according to the information they collect in the system and without any prior negotiation. Because geometric intersections are mutual both intersecting agents attempt to concurrently solve the problem, resulting in over-reactivity.

6.2.2 Swarm suitability test

As the implementation of the final test-bed application approached, it became obvious that using an in-house agent engine has a fundamental drawback: a lack of systematic debugging. Indeed, although the C++-powered agents ran successfully, their implementation was relatively slow because inevitable bugs surfaced in the engine itself. Crashes of applications could not always be accurately attributed to the agent's code or the engine's code. Precious time was therefore being wasted on debugging the engine when efforts should have been focused on implementing agent's mental states and behaviours.

By the time all the concepts of a feature-based agent-driven design system were stabilised, refined and ready for experimentation, viable *off the shelf* alternatives existed to the in-house agent engine. A set of libraries developed at the Santa-Fe institute and called **Swarm** were particularly attractive **[156]**. Swarm is intended to be a useful tool for researchers in a variety of disciplines, especially artificial life. The architecture of Swarm is intended for the simulation of collections of interacting agents. It allows the implementation of a large variety of agent-based models. In the Swarm system the basic unit of simulation is the swarm, a collection of agents executing a schedule of actions. Swarm supports hierarchical modelling approaches whereby agents can be composed of swarms of other agents in nested structures. Swarm provides object-oriented libraries of reusable components for building, analysing, displaying, and controlling experiments on MAS. It is currently available as open source software (free source code provided).



① Normal scheduling in Swarm **② Self scheduling Swarm agents**

Figure 6-7: Scheduling in Swarm applications

It has been systematically debugged, enjoys a growing user base, and offers great opportunities for research and commercial projects. However, there is one main obstacle in adopting Swarm for implementing autonomous agents as defined in 4.2.2. Swarm's activity model is based on schedules that are executed inside each agent community (swarm). Swarm simulates multiagent concurrent activity with a system of activity schedules. A schedule consists of a clock and a mechanism to plan and perform actions inside a swarm according to the value of the clock. Usually, schedules are filled with pre-defined actions (possibly periodic) before launching their execution. This approach is suited to simulations, which represent the core application of Swarm. At first glance, it seems inadequate for fully autonomous agents that require dynamic behaviours to be generated on the fly. However, the

Swarm scheduling libraries are sufficiently flexible to accommodate a potential solution to the problems of agent's autonomy as shown in Figure 6-7.

The normal operating scheme in Swarm application (see Figure 6-7 ①) involves loading schedules with actions during initialisation. Actions are inserted into discrete time steps forming a schedule and can apply to individual agents or group of agents. Actions can themselves be clustered into groups. Both individual actions and action groups can be set to execute periodically. Once the schedule is ready, a local timer is started. For each time step of the timer, the schedule dispatches actions to the appropriate agents for execution. In order to accommodate fully autonomous agents, it is theoretically possible to create a dynamic scheduling scheme in which each agent is responsible for adding actions inside the schedule during execution (see Figure 6-7 ②). When agents detect a situation that requires some action, they can autonomously add an entry into the current schedule, thereby dynamically defining the future activity of the swarm.

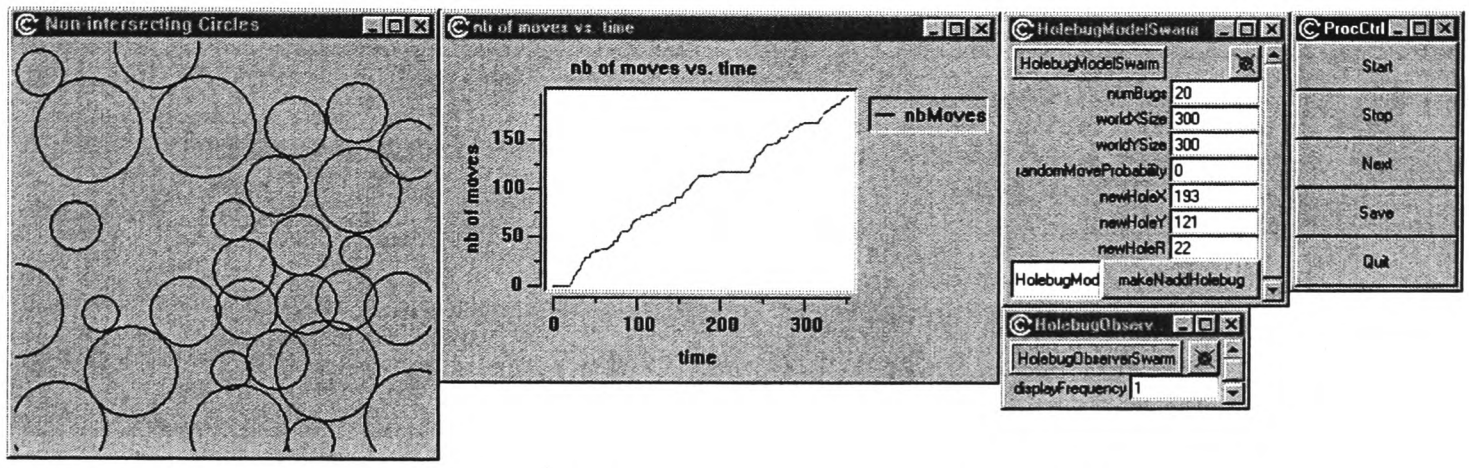


Figure 6-8: Swarm implementation of non-intersecting circles

Figure 6-8 shows a Swarm implementation of the non-intersecting circle test described in 6.2.1.a. It illustrates the valuable tools provided by Swarm (GUI, data collection) and validates the libraries as a suitable framework for the purpose of this thesis. Creating self-scheduling agents has solved the limitations inherent to schedules. These agents can autonomously add or remove items in their schedule at runtime. The community of feature agents uses a non-periodic dynamic schedule to drive its activity. Each agent can access the community schedule, book available time slots and add items to it to perform its activity. This mode of operation scheduling is very similar to that used by most multitasking operating systems. Indeed, when running multiple tasks on a single processor system, the OS slices its processing time into slots that can be dynamically allocated to the running tasks, therefore giving the illusion of parallel processing even though only one process is ever running at one given time. This is exactly what the self-scheduling agents achieve. At any

given time slot, there is only one agent performing its activity, but because they all share the same schedule in an ordered fashion the system behaves like each agent is running in parallel.

6.3 Final Test-bed implementation

This section describes the final MADS implementation realised during the course of this Ph.D. as a proof of concept for the use of autonomous agents to implement design features. The important choice of using the swarm libraries for this system is justified. The objectives and limitations of the realised applications are presented before looking closer at the details of the implementation.

6.3.1 Technological choices

This section explains and justifies the technological choices made for the implementation of the final feature-based MAS. It covers the selection of agents' internal architecture, the adoption of a multiagent engine and an Agent Communication Language (ACL). This section is kept brief since the decision made mostly follow the preliminary experimentation presented in 6.2.

- **BDI motivated agency:** The BDI architecture provides an elegant and flexible way for motivating autonomous agents. The preliminary test described in section 6.2.1.b demonstrated that it was also an efficient method to give several parallel goals to agents.

- **The Swarm simulation libraries:** The schedule-based activity model of Swarm proved to be flexible enough to accommodate autonomous self-scheduling agents (see section 6.2.2). The swarm libraries are aimed at providing a formal framework for computer simulations. As such, it provides efficient data collection mechanisms through specialised objects called probes, which are useful tools for research where observing internal operation inside MAS is as important as the functionality they provide.

- **KQML:** The adoption of KQML as the common ACL is not difficult to justify. As a *de facto* standard in the agent community, it attracts attention more than other languages. Initial tests with KQML demonstrated the flexibility and power of the language. The conversational nature of KQML exchanges makes them intuitively understandable by human being yet allows complex information flow between

agents. Moreover, KQML proves easy to implement and computationally efficient.

6.3.2 Design objectives and limitations

The implementation of the final test-bed application aims to create a system able to assist the user during the design activity. It provides an agent-driven active model, which autonomously analyse its manufacturability and performs self-correction on behalf of the user as described in Chapter 5. It represents a showcase of the potential benefits of applying agent technology to engineering design and manufacturability analysis. Therefore, it is not designed to be useable in real product design. This section defines the objectives, and limitation, of this test-bed.

6.3.2.a Feature-based agent-driven active model

The main objective for the final MADSfm implementation is to obtain a feature-based agent-driven, CAD system capable of modelling machined components of reasonable complexity. The product model produced by the system should be a swarm of autonomous agents instead of a passive data structure. An individual agent must represent each design feature used inside a model. Feature agent must contain sufficient knowledge to detect common manufacturability issues arising from the geometric situation inside the model they inhabit. Feature agents should also be capable of modifying themselves in order to solve at least part of the detected problems. Agents should be motivated to achieve/maintain a manufacturable state inside the model.

MADSfm must eventually demonstrate that feature-agents can be used to assist human beings in the design process. Agents must show an ability to autonomously perform manufacturability analysis on themselves and to perform delegated modifications of the model on behalf of the user. The running CAD applications should provides an environment in which a component's manufacturability is guaranteed during the entire design process.

6.3.2.b 2½D components

The modelling capabilities of the system are limited to purely 2½D components. Salmon strictly defines a 2½D component as one "*for which there is a (one or more) planar surface (called the 'back face') such that the surface normal of any point on the surface of the component, but not on the back face makes an angle of greater than 90° with the surface normal of the back face*" **[62]**.

Manufacturing engineers use the term 2½D to describe components that are easily manufactured on a 3-axis mill without the use of special tools. From a machining point of view a "strictly 2½D" object is made using a 3-axis mill with the additional restriction that the feed axis which refers to the ½ dimension (usually Z) is never used in conjunction with the two others. This allows for the creation of prismatic parts containing only planar and cylindrical faces, and containing only step changes in height.

A 2½D entity can be fully defined by a surface (inside a 2D profile) in the XY plan and a presence interval along Z. The presence interval describes two values in between which the entity exists with a constant 2D cross-section. Entities defined with this representation contain planar surfaces 'horizontally' (whose normal is perpendicular to the XY plan) and any *'vertical'* surfaces (whose normal is parallel to XY). An alternative definition for 2½D components can be proposed based on this implementation scheme. A 2½D entity becomes a geometrical body possessing a constant cross-section, which exists (is present) between two values along the axis normal to its cross-section. 2½D components are made only of 2½D entities sharing the same presence axis.

MADSfm uses this latest representation to implement a unified set of 2½D features. The geometry of all features inside the system are stored as a 2D polygon representing a cross section in the XY plan combined with an interval along Z. Restrictions to the 2D polygons are that it should not be self-intersecting or contain holes. This implementation scheme allows for a unified set of operators to be defined for geometrically manipulating all feature types.

6.3.2.c Machining features

MADSfm uses a limited set of commonly used machining features. They divide into two distinct categories based on their materiality. Positive features represent material used as the basis (blanks) of all designs. Negative features represent the material removal performed through machining operations on positive features. This approach to modelling is described as feature-based destructive modelling (see section 2.4.1).

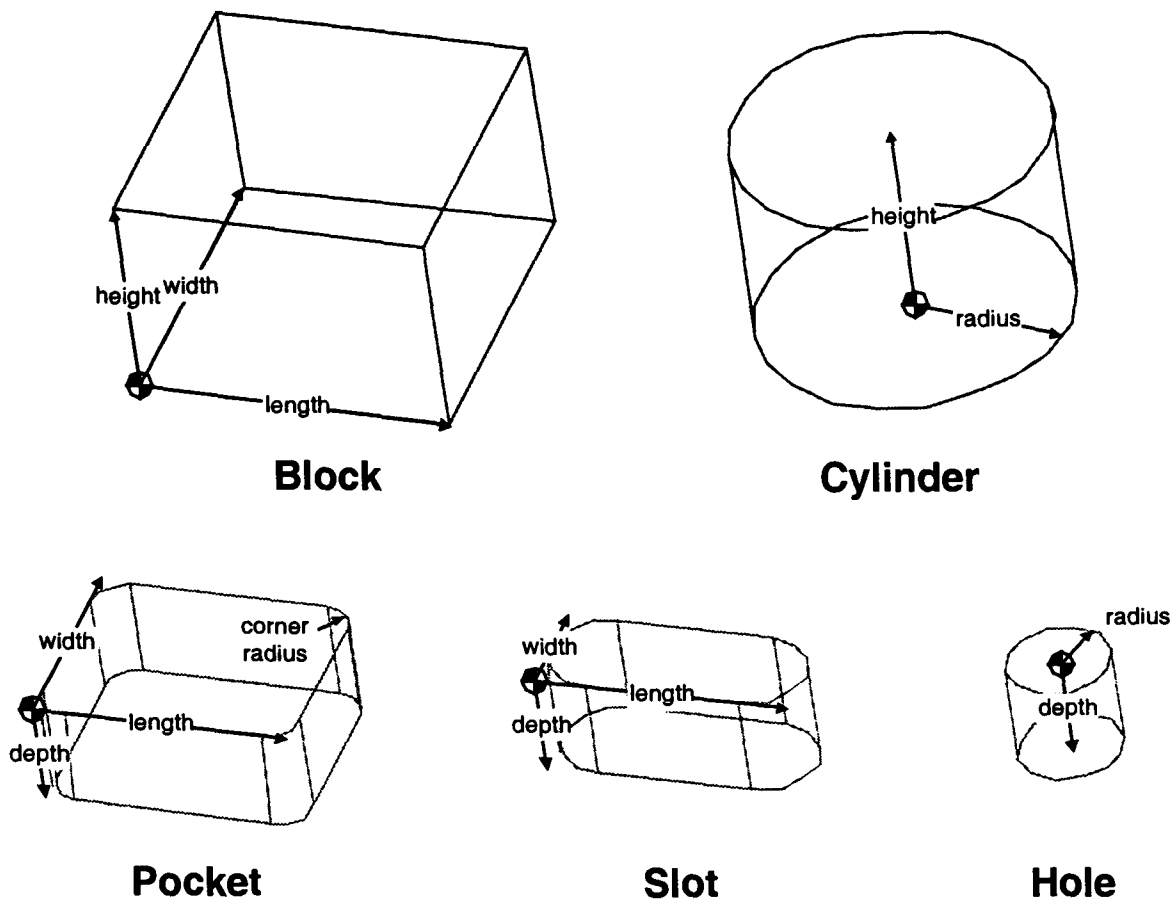


Figure 6-9: MADSfm supported features

Figure 6-9 illustrates the five feature types supported in MADSfm. Blocks and cylinders are the two supported blank shapes. The machining features supported are holes, slots and pockets. This reduced set of feature allows for modelling of fairly complex components and is not considered too restrictive for the purpose of MADSfm. It can be noted that positive features have their origin placed at the bottom (and use a height parameter), while negative feature have theirs placed at the top (and use a depth parameter). This choice is justified because it reflects the physical nature of feature, thereby easing feature positioning. Indeed, consider adding a feature on top of an existing block. Positive features would sit on top of it, while machining features sink into it with their top surface emerging from it.

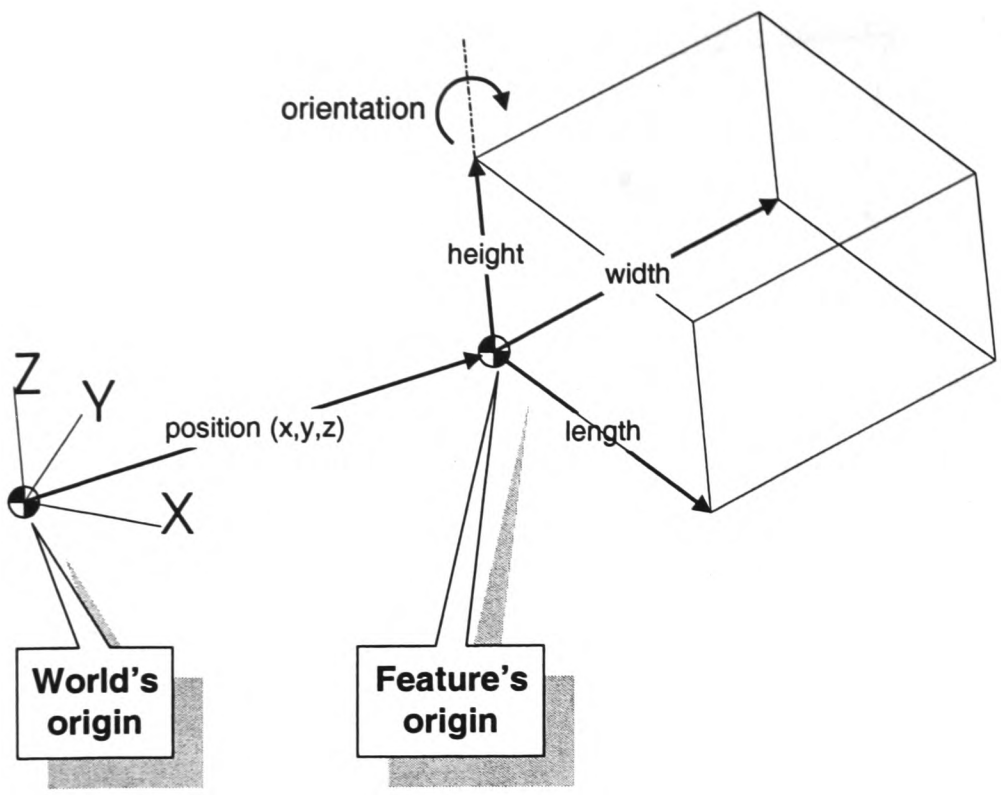


Figure 6-10: Block feature definition example

Each feature is defined by its position, orientation and dimensions. A feature's position, made of three co-ordinates, defines the three-dimensional location of its origin relative to a global origin (see Figure 6-10). Because MADSfm is restricted to 2½D geometry, orientation is fully defined by the angle of rotation along the Z-axis and relative to its origin.

6.3.2.d Validation criteria

One of the objectives of MADSfm is to provide autonomous manufacturing analysis during design. This analysis consists of applying feature validation rules inspired by the work of Vanderbrande **[35]**, Bidarra **[56, 63, 64]** and Gupta **[86]**. These rules represent knowledge about the limitations and simple optimisation of conventional machining processes. By applying such validation criteria, it is possible to detect potential problems with the machining of features. Although providing an incomplete view of manufacturability, they permit the detection of a large proportion of design mistakes. They can prevent the submission to the process planner of designs that are clearly not machinable or sub-optimal by design. The work on feature interaction by Bidarra suggests that other useful criteria could be added to the system such as splitting, disconnection or transmutation **[56]**, but MADSfm is limited to the following 5 criteria.

validation criteria: ① **Pre s ence**

Presence in the finished part expresses the fact that each feature in the model should contribute to at least one surface of the finished part boundaries. This could be also called the

geometric "usefulness" of a feature. Indeed a negative feature placed outside any positive feature will not generate any new surface on the finished part and could therefore be discarded completely.

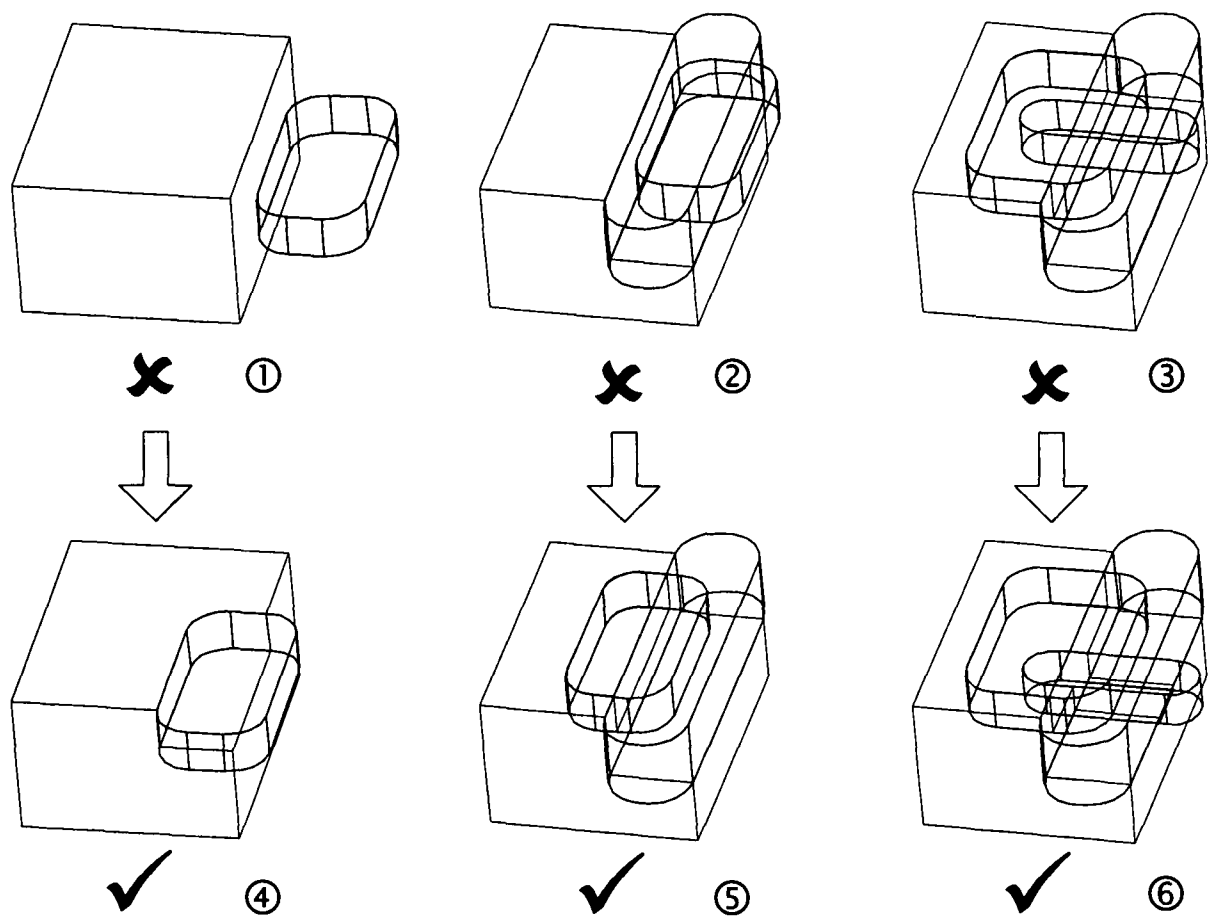


Figure 6-11: Examples of feature presence

Figure 6-11 provides examples of problematic presence configurations in simple components. Two types of configuration exist that creates presence problems for machining features. The most obvious one is the case of a negative feature that intersects with no positive feature (see Figure 6-11①) and is solved by ensuring one or more such intersection exist (see Figure 6-11④). Presence can also be compromised for machining features that do intersect with the blank. If the volume representing the intersection between a negative feature and the blank is fully inside another negative feature (see Figure 6-11②) or the union of several negative features (see Figure 6-11③) it does not contribute to the finished part. Machining features that see their contribution removed by another machining feature become absent from the finished component. From a machining feature point of view, ensuring that the volume representing its intersection with positive features is not fully inside the volume representing its intersection with other negative features can therefore guarantee presence (see Figure 6-11⑤ and ⑥).

Guaranteeing presence inside a finished component thereby involves ensuring presence with at least one positive feature and no *absence relation* with other negative features.

validation criteria: ② **Pro ximity**

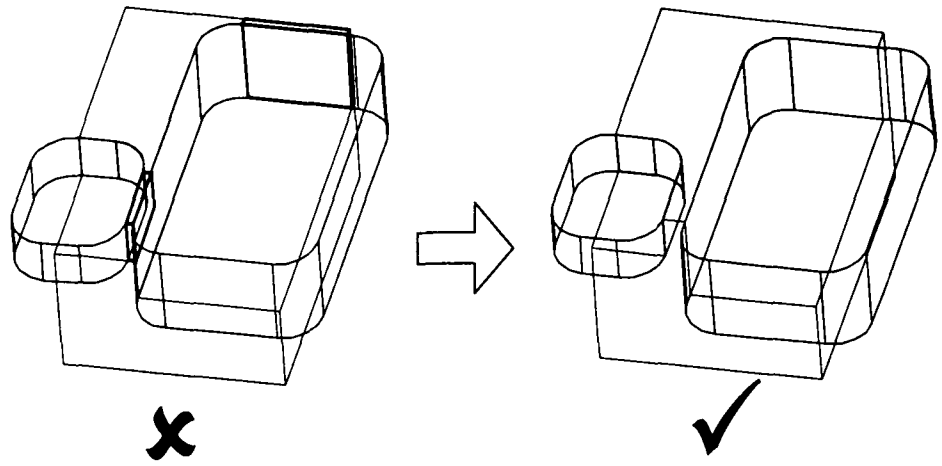


Figure 6-12: Examples of feature proximity

The close proximity of two features, one at least being negative, can generate thin walls of material (illustrated in Figure 6-12) that may be unable to withstand the stress of the cutting process, and bend, or rupture, during machining. Thin wall flexion during machining prevents the achievement of nominal geometry and tolerances. Ruptures have even graver consequences on the production process. The finished part would obviously have to be discarded and cutting tools could also be damaged as a consequence of the thin wall collapse.

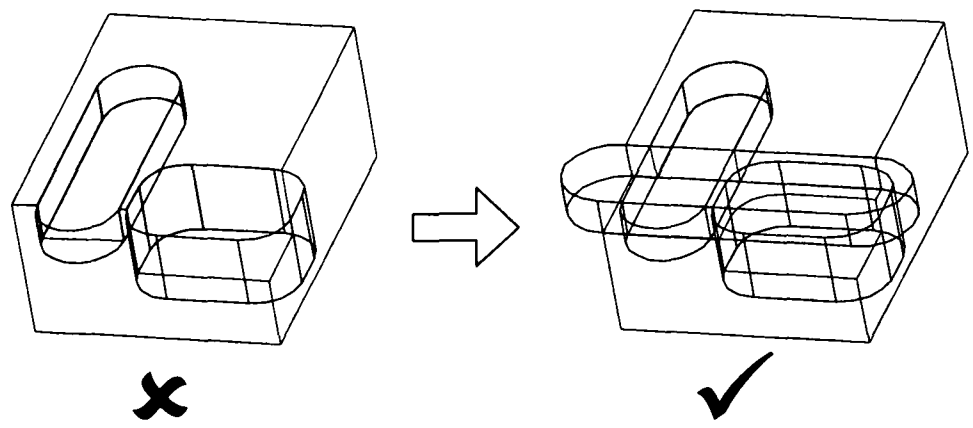


Figure 6-13: Special case of proximity

Two types of proximity problems can occur in MADSfm. Thin walls might be created between two negative features or between a negative feature and the boundaries of positive features. Both are illustrated in Figure 6-12. Feature-based designs can be validated against thin walls by ensuring minimum clearance between features. It should be noted that cases exist where simply measuring inter-feature distances might wrongly detect proximity problems. Such a case is shown in Figure 6-13.

Two negative features are placed closely together, thereby creating a thin wall between them (see left). This thin wall can be detected by measuring the inter-feature distance.

However, if another negative feature is added that intersect with the thin wall mentioned (see right), the distance measure will still detect a thin wall despite its absence from the finished component.

validation criteria: ③ **Acc ess**

In order to be machined, a machining feature has to be accessible to a cutting tool. Access is a very important criterion, and can be used to detect and resolve a large number of non-manufacturable designs. MADSfm makes a deliberate assumption that all negative features must be accessible through their Z-axis. In the case of strictly 2½D components to be machined without special tooling (which excludes T shaped cutter), this restriction can be justified.

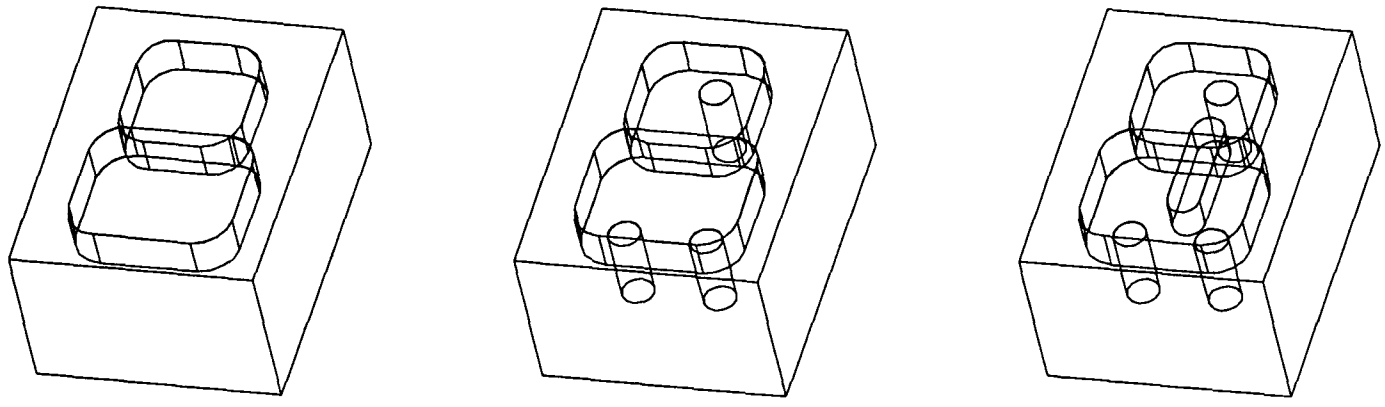


Figure 6-14: Examples of feature access

Access can be obtained either directly when the feature emerge to open space (with the right orientation) or indirectly through another accessible feature. *Direct access* describes a feature, which can be accessed directly by the cutting tool. Geometrically, features whose top face is not below any positive feature have a direct accessibility (see the two pockets in Figure 6-14①). Indirect access can be further divided between full access and partial access. *Full indirect access* describe a feature obtaining indirect access through a single other feature. Its 2D profile is thereby fully enclosed inside the 2D profile of the other feature (see the three hole in Figure 6-14②). *Partial indirect access* describes a feature obtaining indirect access through more than one other negative feature. In that case, no single feature can provide an acceptable route to the feature. Several features provide a partial access and the union of these allows tool access to the feature (see the slot in Figure 6-14③).

From a machining point of view, indirect accesses represent precedence rules between features. A feature with full indirect access must be machined after the feature providing the access route. A feature with partial indirect access must be machined after all features

contributing to its access route. In some cases, a feature might possess several indirect accesses. Such features can be machined after at least one indirect access has been created through machining.

validation criteria: ④ Co l lision

Collisions are geometric configurations of design features that are either physically impossible (e.g. intersection of positive features) or that could prevent machining of the part. Collisions between features are not practical to classify and often belong to "special cases".

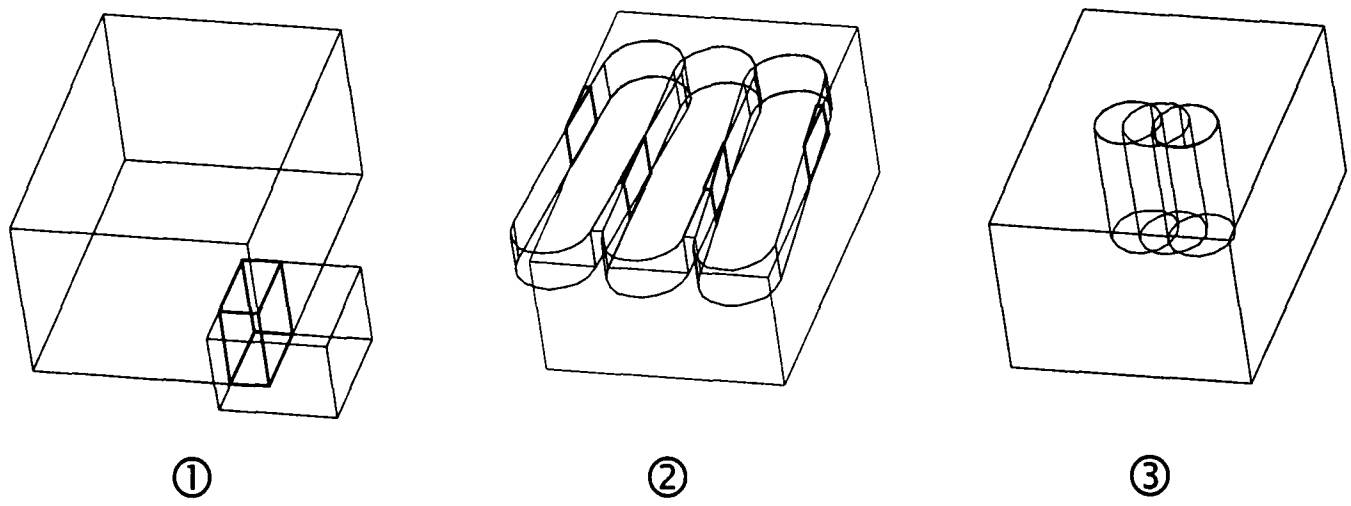


Figure 6-15: Examples of feature collision

An obvious collision occurs when two positive features intersect (see Figure 6-15①) which translates into a physical impossibility. Less apparent collisions can also appear in a model that are related to limitations of the machining processes used to physically create negative features. Examples are machining features intersecting inside the blank (or with the blank boundaries) and creating acute angle in the finished part (see Figure 6-15②). This type of collision is clearly related to the thin wall problems described earlier. However, they are more difficult to detect and automatic solutions are not readily available. A third example is given that shows three intersecting holes (see Figure 6-15③). This array of holes would not be possible to manufacture because of their intersection. Creating the first hole would deny lateral support to the drill for the second and third drilling operations. It can be argued that this is clearly an erroneous design. Yet, automatic detection of such problems during design could save valuable time.

validation criteria: ⑤ Mi n imality

Minimality is an issue addressed by Gupta's automatic manufacturability analysis of machined parts **[86]**. It relates to the truncation of machining features in order to optimise the cutting process. Basically, it requires the identification and removal of areas in a design that

would be "*machined twice*" from the blank. Obviously, it is not physically possible to remove the same area twice. However, valuable time can be lost by moving tools at *slow* cutting speed through areas that have already been machined and could thereby be covered at faster non-cutting speed. It also involves the truncation of features that machine the "air" so as to be the smallest feature that achieves the finished shape

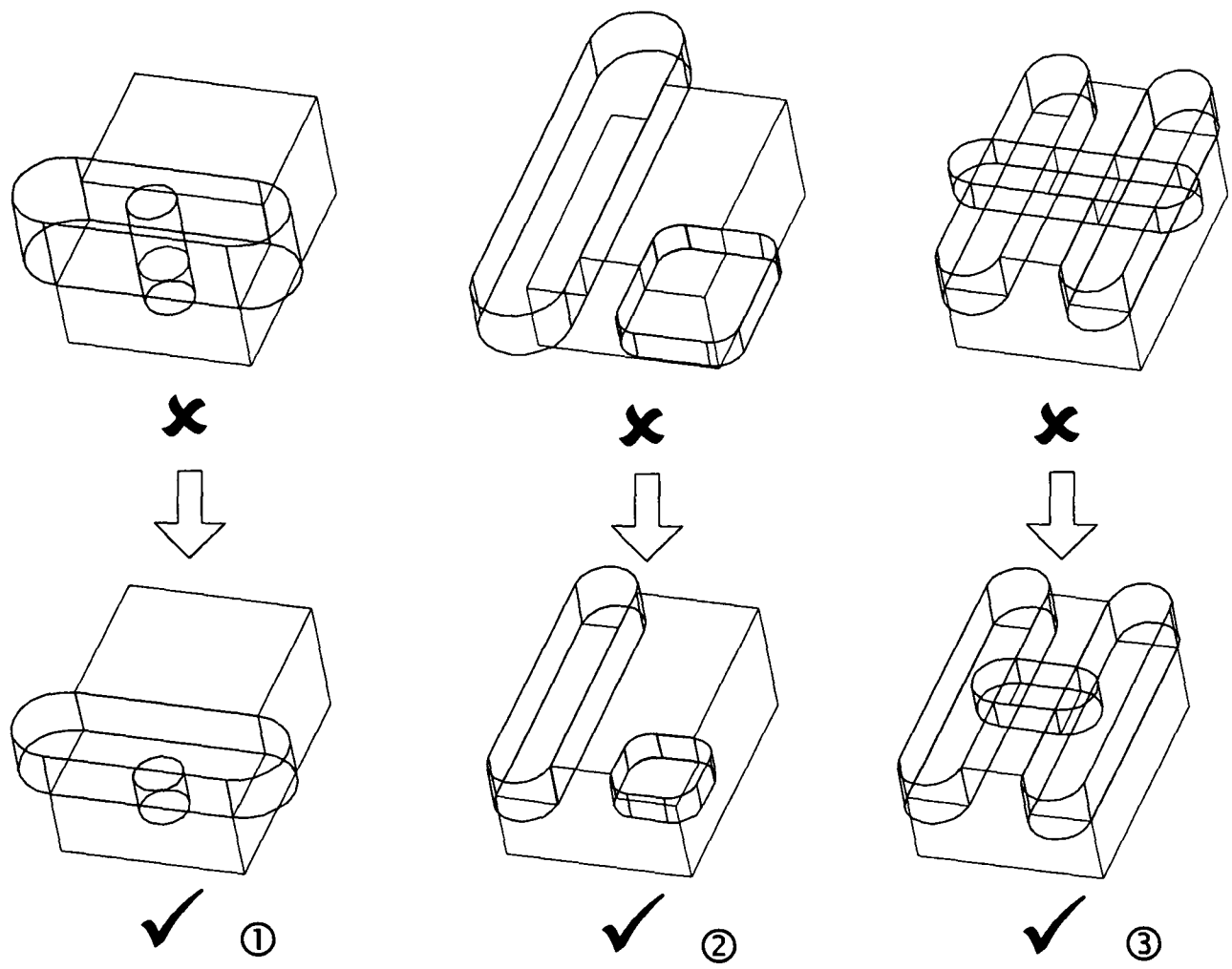


Figure 6-16: Examples of feature minimality

In strict 2½D environment, feature minimality can be considered almost separately in the XY plan and along the Z-axis. Z minimality describes the fact that a feature is just "tall" enough to produce its contribution on the finished part (see Figure 6-16①). Z minimality of machining feature can reduce machining time by ensuring the optimal depth of feature, which should translate into an optimal depth of cut. It should be noted that minimising features along Z might reduce flexibility in process planning by introducing precedence rules between features (see Figure 6-16①). Minimality is also an issue in the XY plane. One XY minimality configuration is a feature partially intersecting with the blank (see Figure 6-16②). Parts of such features do not intersect with the blank, which means that not all of the machining cycle used will actually remove material. It is possible to maximise effective machining time by reducing these areas to the smallest possible (see Figure 6-16②). Note that in the case of slots used to create steps, this truncation might be performed only on the

slot's length since its width might have been chosen as a specific tool's diameter. If it is not the case, the width can also be minimised. Finally, XY minimality can involve overlapping machining features (see Figure 6-16③).

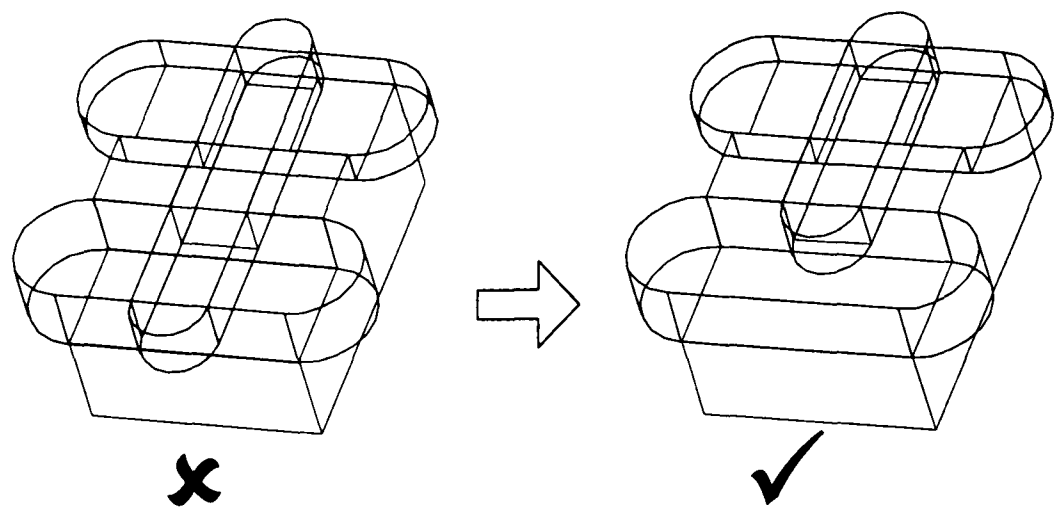


Figure 6-17: XY minimality depends on Z overlap

In some cases, it is possible to truncate such features and remove unnecessary shared volume. However, as demonstrated in Figure 6-17, the removal of shared volumes by XY truncation can only be done if a full Z overlap exist between features.

6.3.3 System Architecture

The general architecture of MADSfm is presented in this section. An overview of the internal organisation of the system is given that shows the relationships between all its constituting components. The function and implementation of the different elements of MADSfm is subsequently explained in more details.

6.3.3.a Overview

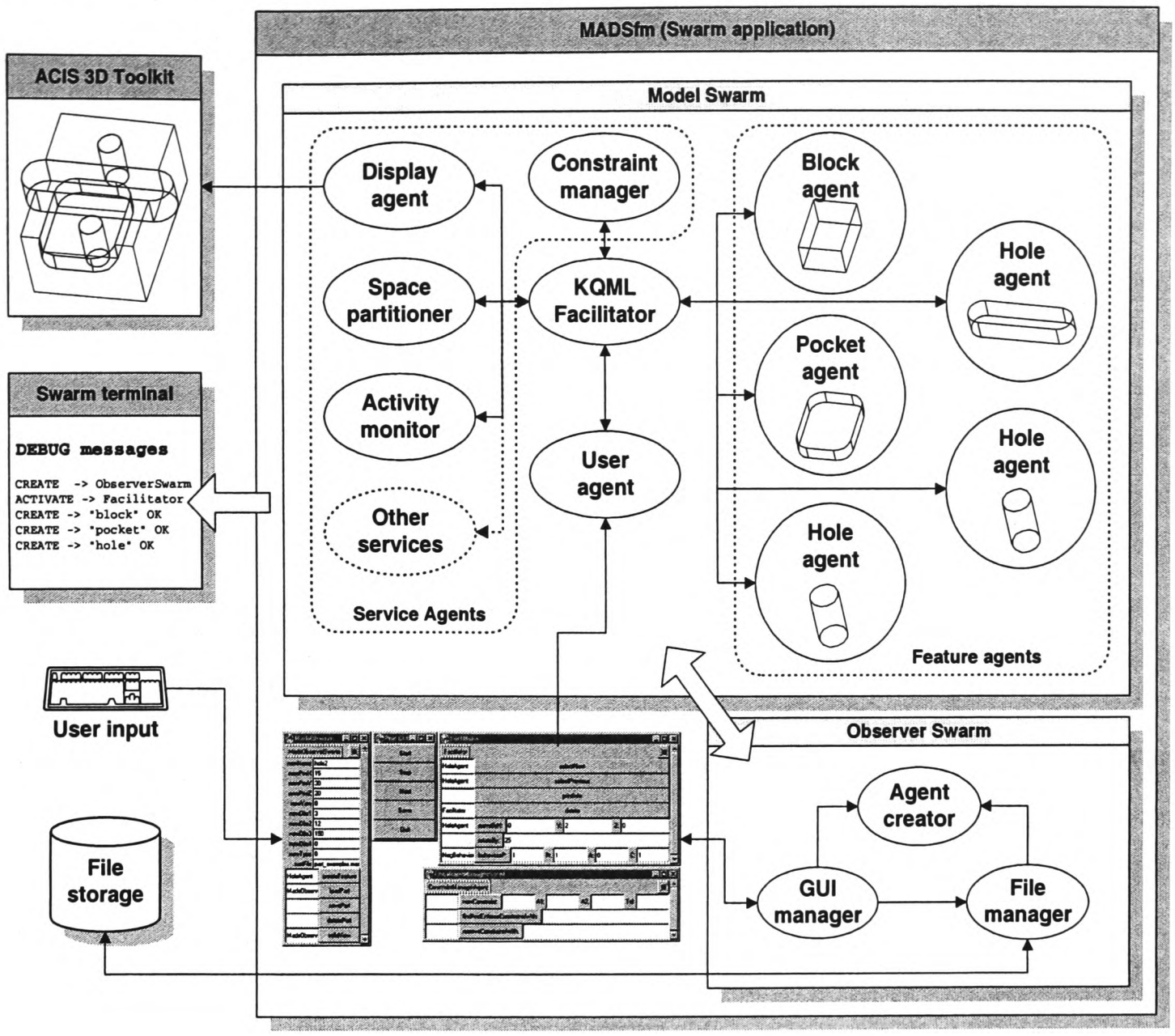


Figure 6-18: MADSfm architecture overview

MADSfm is a complex Swarm application supporting several types of autonomous agents working together to provide 2½D modelling capabilities with automatic manufacturability analysis and solving. Figure 6-18 shows an overview of the global architecture of the system. The main swarm application is made of two connected swarms. The *observer swarm* is a top-level activity group used to provide the interface between the core agents of MADSfm and the system's environment (e.g. user and other applications). Observer swarms are a common occurrence in Swarm application and usually provide display (GUI) and data collection capabilities. The observer Swarm used in MADSfm is not dissimilar to conventional ones. A GUI manager generates the controls that are needed for user input while a file manager handles the loading and saving of designs to and from files. However, MADSfm is not a traditional Swarm simulation because it also contains an agent creator used to dynamically populate the *model swarm* with new agents in response to user input. The *model Swarm* represents the operating core of the system. It is where all autonomous activity takes place,

which results in the system's usefulness. Several types of agents co-exist inside the model swarm. They can be divided in three categories. *Feature agents* represent individual features making out the design mechanical component. They are responsible for ensuring their manufacturability during design. *Service agents* are non-geometric agents dedicated to providing high-level services to the agent community. These include a display agent for 3D visualisation, a space partitioner for ensuring locality of communications, an activity monitor to detect livelocks and a constraint manager to apply basic geometrical constraints to features. Service agents are not a strict requirement for MADSfm proper operation but represent modular functional entities, which can be added to enhance the system's capabilities or performance. All service agents are discussed in more details later in this chapter. A third category of agents groups all agent types in the model swarm that do not fit in the first two categories. It contains non-geometric agents that are essential to MADSfm's activity. The *KQML facilitator* is vital to inter-agent communication. It holds the conversion table between symbolic agent names and their physical addresses. It also provides messaging services such as registration, message forwarding and broadcasting. The *user agent* is an empty agent shell used to give MADSfm' users the possibility to converse with other agents on a peer-to-peer basis.

The user of MADSfm interfaces in three different ways with the system. The GUI provided by the GUI manager, acts as the only input mechanism. It supplies the necessary input fields (type, position, orientation and dimensions) for creating new features. Geometric manipulations (translation and rotation) are available through specific fields and buttons. Behavioural properties of agents can also be manipulated through MADSfm's GUI. Specialised controls permit predefined behaviours corresponding to each validation criteria described earlier (see section 6.3.2.d) to be activated or disabled. The GUI also give access the global swarm activity by allowing schedules to be started, stopped or executed step by step. A second link with the system is provided by messages displayed on a Swarm console (or terminal). Agents inside the system use the console to display status messages that the user can use to appreciate current activity. Finally a dedicated service agent provide visual feedback by displaying a 3D representation of feature agents inside an ACIS window.

6.3.3.b Feature agents

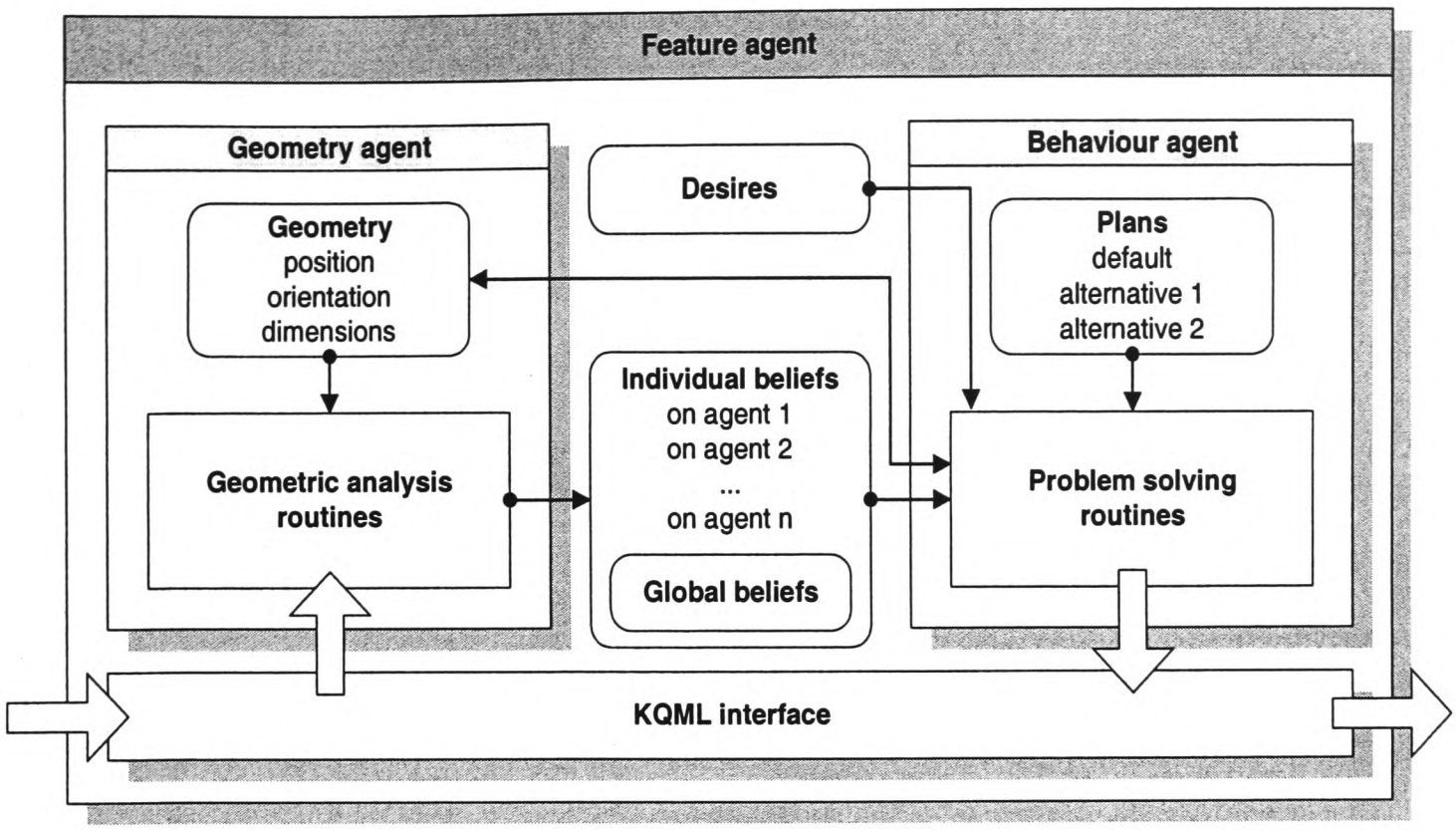


Figure 6-19: MADSfm internal agent architecture

Feature agents are arguably the most important entities inside MADSfm. They represent the designed component and are responsible for most of the activity inside the system. In that respect they deserve particular attention. Figure 6-19 illustrates their internal architecture, which reveals the duality of their activity. Indeed each feature agent is made of two distinct autonomous agents performing tasks independently from one another. A geometry agent performs geometric analysis on incoming feature data and generates high-level beliefs about known feature agents. A behaviour agent uses these beliefs to assess the feature's manufacturability according pre-defined validation rules (see section 6.3.2.d) and uses solving routines to eliminate detected problems.

Feature agents communicate with other agents through their KQML interface. This interface handles the complexity of KQML formatting and addressing. It filters unwanted messages and strips supported messages of unnecessary information before passing their content to the geometric agent for analysis. It also generates KQML fields before sending out messages generated by the behaviour agent.

Feature agent architectur e : ① **Geometry agent**

The geometry agent holds the geometric information defining the feature, such as feature type, position, orientation and dimensions. It also contains the geometric analysis routines used for building beliefs about other features in the system. The geometric agent

encapsulates the specific implementation of geometry and generates high-level, representation-independent, beliefs. This separation offers a degree of independence from geometric representation of features.

Feature agent architectur e : ② Behaviour agent

The behaviour agent contains the manufacturability detection and solving knowledge of features. It contains a database of pre-defined plans, a mechanism for selecting appropriate plans and for executing them. It is important to notice that it is independent of the low-level geometric representation used by the system. It relies fully on the dynamic beliefs generated by the geometric agent to perform its activity.

Feature agent architectur e : ③ Desires and Beliefs

Desires and beliefs belong to the feature agent and are shared between geometric and behavioural activities. Features' desires and beliefs are pre-defined and paired to represent the validation criteria used by features (presence, proximity, collision, access and minimality). They represent the manufacturability status of a feature. Desires define acceptable values for beliefs from the manufacturing point of view. A positive feature representing a workpiece holds the following desires.

```
PRESENCE > 0        ensures contribution to finished part
PROMIXITY = 0       eliminates thin walls
COLLISION = 0       eliminates collisions
```

A negative feature possesses desires that capture the limitations of the machining process.

```
PRESENCE > 0        ensures contribution to finished part
PROMIXITY = 0       eliminates thin walls
COLLISION = 0       eliminates collisions
ACCESS > 0          ensures tool access
MINIMALITY = 0      ensures feature's minimality
```

Geometric agents generate two types of beliefs. *Global* feature beliefs have just been presented and reflect the manufacturability of the feature relative to its entire environment. *Individual* beliefs are also generated that represent the relation between a feature and other features individually. To avoid confusion between individual and global beliefs, all individual beliefs have a name starting with underscore ("_"). The individual beliefs supported by MADSfm are the following:

```
_PRESENCE            feature grants presence on finished part
_ABSENCE             feature fully denies presence
_ABSENCE_PARTIAL     feature partially denies presence
_COLLISION           feature collides
_PROXIMITY           feature is closer than pre-defined clearance

_ACCESS_ALLOW              feature grants tool access fully
_ACCESS_DENY               feature denies tool access fully
_ACCESS_PARTIAL_ALLOW      feature grants tool access partially
_ACCESS_PARTIAL_DENY       feature denies tool access partially
_MINIMALITY                feature creates minimality issue
_MINIMALITY_PARTIAL        feature partially creates minimality issue
```

The geometric agent analyses incoming geometric information about other features and generates beliefs abstracted from the specific geometric representation used. Individual beliefs are generated, which correspond to the feature validation criteria described in 6.3.2.d. These individual beliefs are Boolean values reflecting one to one relations with each known feature. For example, if the geometric elements generated for a feature give no XY intersection, a perfect overlap along Z and a smallest distance smaller than a pre-defined thin wall thickness, a belief of _PROXIMITY will be activated.

Geometry agents use the individual beliefs for each known feature and compile them into global beliefs that reflect the global situation of the feature it represents. _PROXIMITY, _COLLISION and _MINIMALITY beliefs need only be counted to generate their global counterparts. Individual _ABSENCE and _PRESENCE beliefs are combined into a global PRESENCE belief. Combining the four individual access beliefs also creates a global ACCESS belief. This aggregation of individual beliefs into global beliefs creates manufacturability status of the feature corresponding to the validation rules described in 6.3.2.d.

6.3.3.c Service Agents

Service agents are non-geometric agents providing high-level functionality to other agents. They are not usually essential to the operation of the MAS but increase its capabilities and performance.

Service agents: ① **Displa y agent**

Apart from its GUI controls (windows, buttons and text fields), Swarm's display capabilities are limited to 2D widgets common to most windowing systems (lines, circles, polygons, text, etc.). This is obviously not satisfactory when trying to display 2½D geometry on screen. Therefore, an external rendering is used to provide visual feedback to the user concerning the component's geometry. The selected tool; the ACIS® 3D Toolkit by Spatial

technology inc. is a Scheme interpreter bolted on top of the powerful ACIS® 3D kernel. It combines the power of a full 3D kernel with the ease of use of a command line interpreter. A display agent is thereby added to MADSfm that sends Scheme commands to the ACIS® 3D toolkit to provide a full 3D view of the designed component.

Service agents: ② Constr a int manager

In mechanical design, it is often necessary to define geometric constraints between entities inside a model. This is particularly true when designing using conventional CAD methods but it also applies to feature-based design. An elementary constraint manager agent is thereby added to MADSfm that allows simple constraints to be imposed on features. The constraint manager supports the creation of bi-directional constraints between features and ensures their evaluation and relaxation during geometric changes. Concentricity and orientation constraints can be expressed. This represents very limited support for geometric constraints but demonstrates the feasibility of constraint propagation inside a multiagent CAD system.

Service agents: ③ Space partitioner

The principle of locality is a well-understood concept both in agent technology and feature-based modelling. Locality is recognised as an essential part of agency **[104]**. Indeed it is the ability to take local decisions based on local knowledge that makes MAS such a flexible and powerful systems. Agents rely on their localised activity and emergent behaviour to replace global knowledge and control used in conventional approaches.

Machining features used in mechanical design also involve the principle of geometric locality. Vancza and Marcus argue that "*most changes of the [model] have usually only moderate, local consequences. [...] According to the locality principle, the [process] planning domain should be partitioned into regions and the effect of actions should be kept within strict bounds*" **[48]**. This principle is confirmed by the validation criteria used in MADSfm, which express local requirements concerning geometric interaction between features.

The space partitioner agent is a performance enhancing entity inside MADSfm. As such, its activity is completely transparent to the user. It is used to keep track of a feature's geometric locality inside the system. It uses these locality properties to dynamically filter broadcast messages between features, thereby reducing the overall communication load. The KQML facilitator delegates the task of performing broadcasts to the space partitioner. The

later only forwards broadcast messages to features in the sender's locality, instead of automatically forwarding to all features.

Service agents: ④ **Activit y monitor**

The activity monitor agent is another entity whose activity in not necessarily obvious to the user. It monitors the global activity of the agent community and performs useful actions depending on the state of the system. Its main use in MADSfm is to detect potential livelocks between features and attempt to solve them automatically.

6.3.3.d KQML facilitator

The KQML facilitator is the most active entity inside MADSfm. Its duty is that of a central post office, providing the infrastructure for inter-agent communication. In particular it holds the central address book for all agents. The address book is a table containing the symbolic name and physical address of all registered agents inside MADSfm. The KQML facilitator maintains this table and uses it to provide physical delivery addresses for KQML messages.

The KQML facilitator offers various messaging services to the agent community. The first service is to allow new agents to register with the KQML facilitator so making them reachable by all other registered agents. Note that agents leaving the system must unregister to avoid corrupting the address book. Two more services are offered that handle message delivery, namely broadcasting and forwarding. In broadcasting, the facilitator sends copies of a given message to all registered agents in the system. In forwarding, it redirects a given message to its rightful receiver.

The KQML facilitator is a simple but vital entity for MADSfm. It supports a subset of KQML performatives (register, unregister, broadcast, forward) that can be used to offer high-level communication services.

6.3.3.e User agent

The user agent is not really an autonomous agent. In fact, it is the empty shell of an agent used to put the human user of MADSfm "inside" the agent community. It provides the user with a working KQML layer that can be used to send messages to other agents in the system. In particular, the user agent is used to send requests to feature agents. When the user uses

MADSfm's GUI to modify existing features, his input is actually routed through the user agent, translated into an appropriate KQML message and sent to the feature agents.

6.3.4 System Operation

The general operation of the MADS is now described. The system architecture discussed in section 6.3.3 is an extremely modular one and each agent can be discussed separately. Feature agents, in particular, are based on a total separation of geometry and behaviour. Therefore their operation within that architecture can also be discussed separately from both point of views.

6.3.4.a Feature Agents

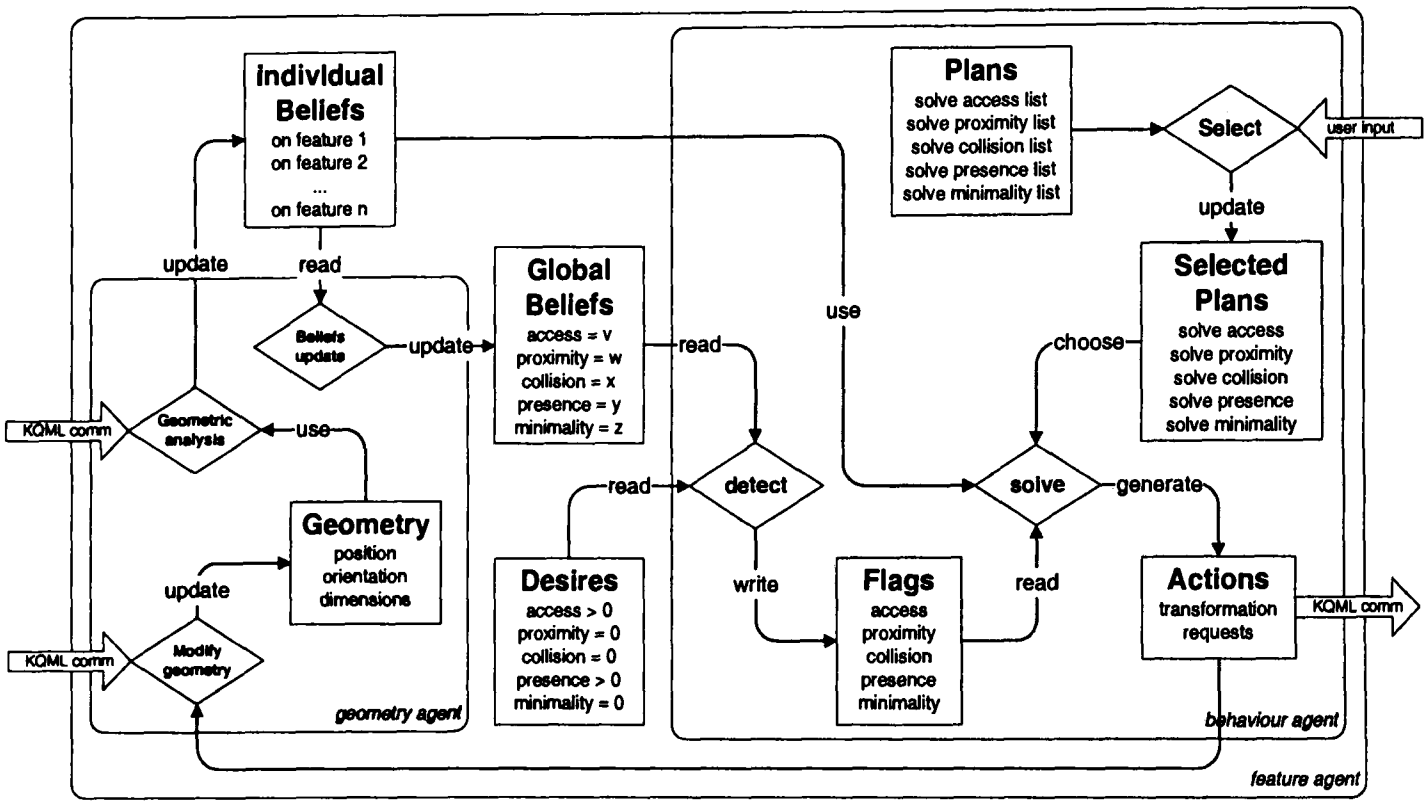


Figure 6-20: Operation of a feature agent

Feature agents account for most of the activity inside MADSfm. The division between geometric and behavioural activity is evident in Figure 6-20. Two independent agents carry out their duties in parallel inside each feature. Geometry agents hold and maintain the geometric definition of features. They are also in charge of analysing the geometry of other feature agent and generating beliefs, which form a feature's awareness of its environment. Behaviour agents enforce the manufacturability validation rules for the features that they represent. They compare the dynamic beliefs and pre-defined desires in order to detect manufacturability problems. The solving capabilities of behaviour agents are contained in a database of pre-defined strategies. Their application relies on beliefs generated during

geometric analysis and result in geometric transformations, which are mostly performed on the feature itself but can also, be requested of others inside the system.

The operation of coupled geometry and behaviour agent is completely separate. Such separation is important because it makes geometric analysis independent from other feature activities. This allows the geometry agent to keep beliefs up-to-date regardless of behavioural activity. More detailed descriptions of the dual activity of feature agents is given in sections 6.3.4.b and 6.3.4.c.

6.3.4.b Geometry Agents

A geometry agent holds and maintains all the data that defines a feature in conventional 3D systems. Position in space, dimensions, parameters and materiality are part of the geometric agent's internal data structures. It is important however to notice that the 3D data is now owned by the feature itself. Indeed, it is the feature itself, through its geometry agent that ensures data consistency. But the geometry agent handles more than the feature's geometric data and also builds and maintains a number of beliefs about its current status as well as a local snapshot of the world it is living in.

The main task performed by the Geometry Agents is to process all the geometric data that is sent around their local swarm and to use this data to maintains a set of coherent beliefs that can be used by the Behaviour Agents. Typically, geometry agents receive notifications of changes in the component's geometry. When a geometry agent receives a notification from another feature, it collects the new geometric data from it and processes it to update its internal beliefs. If a change is made to the beliefs as a result of the geometric changes, the agent sends a message toward its respective behaviour agent so that potential actions can be investigated in the light of the new situation.

The task of maintaining beliefs is the most demanding one from the computing resources point of view. Indeed, each notification of change inside the component triggers a series of computations that check for any potential effects on the current beliefs. Using conventional geometrical algorithms, this involves testing Z overlap, and 2D XY profiles for intersections, minimum distances and minimum angle at intersections. It also requires the (re)generation of individual and global beliefs for each received geometric message.

The geometry agent runs concurrently with its coupled behaviour agent therefore ensuring that beliefs are always coherent with the reality of the component. It endlessly updates itself

in order to provide the most complete representation of the world that its behavioural counter-part can use to make efficient decisions for the benefit of the feature that it represents. Geometry agents perform three distinct activities:

Geometry agents activity: ① **G eometric evaluations**

The KQML layer of feature agents is in charge of extracting geometric data from received messages. The extracted geometry represents the 2½D definition of a feature and is stored as a 2D XY profile and a Z interval as described in 6.3.2.b. The geometric evaluations consist in testing the incoming feature geometry against its own and extract useful facts about mutual spatial relationship.

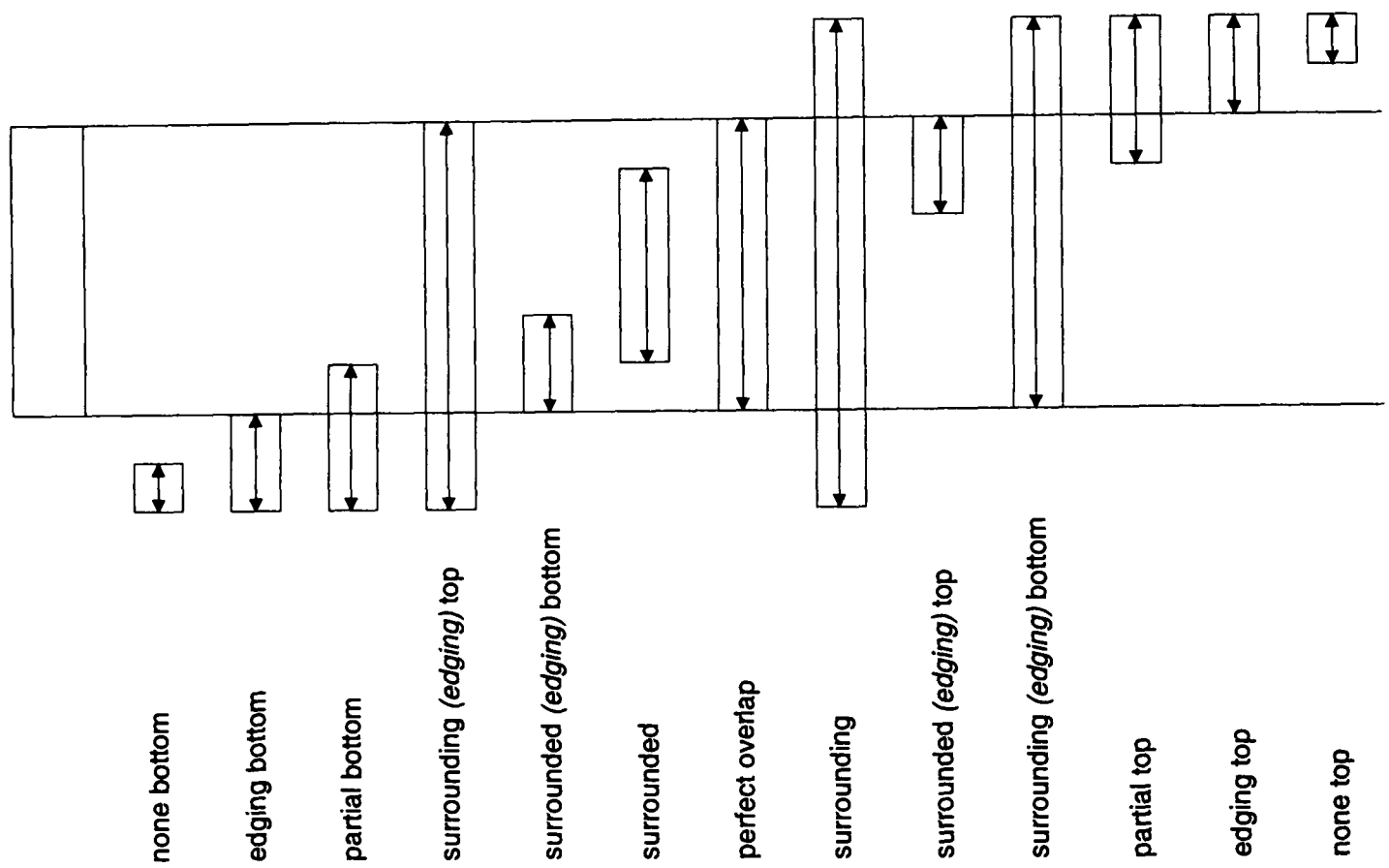


Figure 6-21: Inter-feature Z intersection types

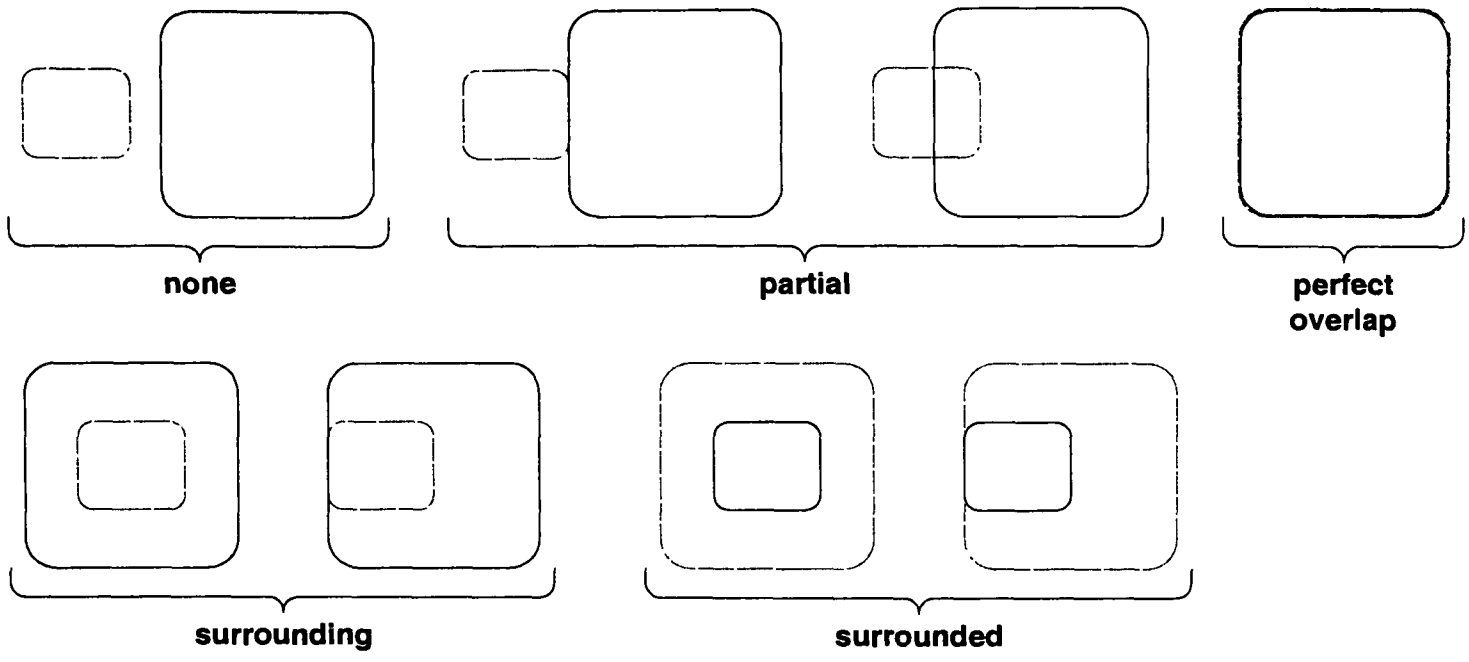


Figure 6-22: Inter-feature XY intersection types

The calculated facts include intersection types in the XY plan, intersection along Z (see Figure 6-21 and Figure 6-22), smallest inter-feature distance, smallest inter-feature angle, and more. The calculation of all these geometric elements is very important. The results obtained are stored by the geometric agent inside the belief database and are used for belief update but also by the behaviour agent during problem solving.

Geometry agents activity: ② **B elief updates**

Once geometric analysis is done and all geometric elements have been stored, the geometric agent can deduce individual beliefs concerning the incoming feature and global beliefs concerning the feature it represents inside the model.

Individual beliefs are obtained by checking the geometric elements extracted during analysis and applying pre-defined rules. By default all individual beliefs are set to *false* and rules consist in condition for setting them to *true*. Different rules are applied depending on the materiality of both the features as shown in the following tables.

XY intersection type	Z intersection type[6]	Smallest distance	Smallest angle	Individual belief set to *true*
≠ none	≠ none bottom ≠ none top ≠ edging bottom ≠ edging top	--	--	_COLLISION

Table 6-2: Rules for positive features assessing positive features

[6] Multiple (=) are logically OR-ed, and multiple (≠) are logically AND-ed

XY intersection type	Z intersection type[6]	Smallest distance	Smallest angle	Individual belief set to *true*
≠ none	≠ none bottom ≠ none top ≠ edging bottom ≠ edging top	--	--	_PRESENCE
= surrounding	= none top = surrounding = surrounding bottom = partial top = edging top	--	--	_ACCESS_DENY
= partial	= none top = surrounding = surrounding bottom = partial top = edging top	--	--	_ACCESS_PARTIAL_DENY
≠ none	≠ none bottom ≠ none top ≠ edging bottom ≠ edging top	< min	--	_PROXIMITY
≠ none	≠ none bottom ≠ none top ≠ edging bottom ≠ edging top	--	< min	_COLLISION

Table 6-3: Rules for negative features assessing positive features

XY intersection type	Z intersection type[6]	Smallest distance	Smallest angle	Individual belief set to *true*
= surrounding	= surrounding = surrounding bottom = surrounding top = perfect overlap	--	--	_ABSENCE
= partial	= surrounding = surrounding bottom = surrounding top = perfect overlap	--	--	_ABSENCE_PARTIAL
= none	≠ none bottom ≠ none top ≠ edging bottom ≠ edging top	< min	--	_PROXIMITY
≠ none	≠ none bottom ≠ none top ≠ edging bottom ≠ edging top	--	< min	_COLLISION
= surrounding	= partial top = edging top	--	--	_ACCESS_ALLOW
= partial	= partial top = edging top	--	--	_ACCESS_PARTIAL_ALLOW
= surrounding	= partial top	--	--	_MINIMALITY
= partial	= partial top	--	--	_MINIMALITY_PARTIAL

Table 6-4: Rules for negative features assessing negative features

The application of the rules listed in Table 6-2, Table 6-3 and Table 6-4 yields the individual beliefs concerning the feature broadcasting its geometry. However, there are special cases that must be dealt with before the geometric agent starts generating global beliefs. These special cases concern all the partial contributions of the received feature. These include partial *access*, *absence* and *minimality*. The three are similar and ensue from the possibility of several machining features combining their partial interaction to create a full one. The geometric agent must further analyse all partial beliefs in order to determine whether they do combine with other partial beliefs to provide a complete one. All partial contributions are resolved in a similar manner.

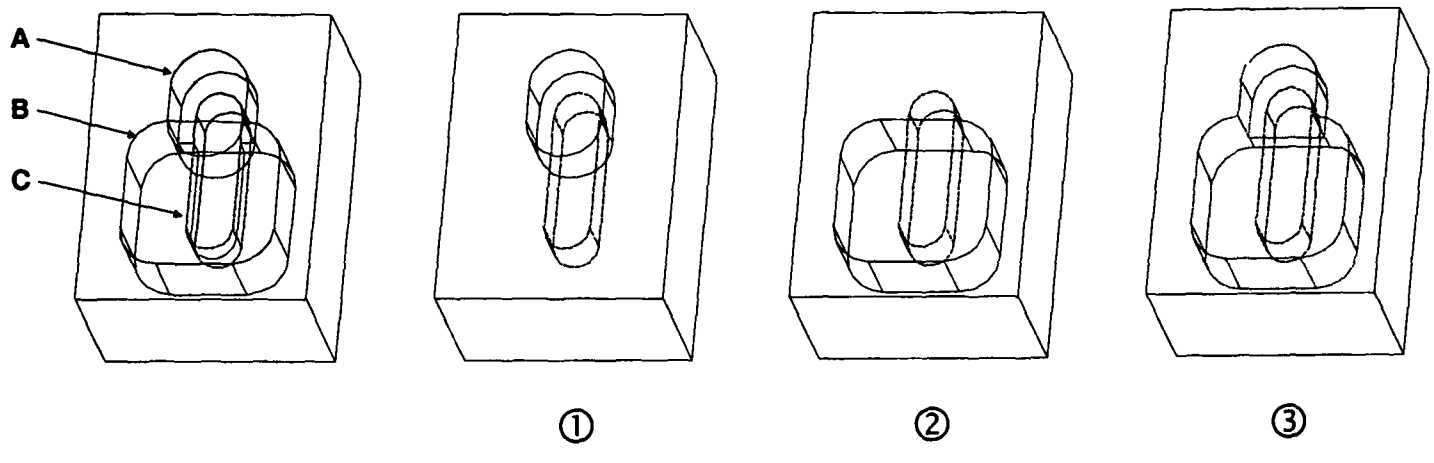


Figure 6-23: Example of partial contribution to full tool access

To simplify, let's consider partial access and use Figure 6-23 as an example. If feature agent C receives a broadcast from feature A, the _ACCESS_PARTIAL_ALLOW belief will be set for it after geometric analysis (see Figure 6-23 ①). The geometric agent has to determine whether this _ACCESS_PARTIAL_ALLOW plays a role in providing full access to feature C. For this, it checks its belief database for existing _ACCESS_PARTIAL_ALLOW with other known features. It founds that feature B also set a _ACCESS_PARTIAL_ALLOW (see Figure 6-23 ②). A new access check is thereby performed between the geometry of feature C and the union of features A and B. It is found that the union A+B (see Figure 6-23 ③) fully grants tool access. The _ACCESS_PARTIAL_ALLOW belief is therefore maintained as a belief concerning A. This individual beliefs of C marks the fact that A's partial access actually contributes to providing full tool access when combined with other features. If A+B had not granted full access, the _ACCESS_PARTIAL_ALLOW would have been removed from the belief concerning A.

Dealing with partial contribution is by far the most expensive part of the belief update. It requires belief retrieval, complex geometric union operations and additional geometric tests for access, absence or minimality to be performed.

The final stage of belief update is to generate global beliefs reflecting the situation of the feature relative to its entire environment. Combining all individual beliefs contained inside the agent produces five global counters. The rules use to generate them are the following:

```
PROXIMITY = Σ(_PROXIMITY)
COLLISION = Σ(_COLLISION)
MINIMALITY = Σ(_MINIMALITY)
ACCESS = If (Σ(_ACCESS_ALLOW) > 0) Σ(_ACCESS_ALLOW)
         Else 1 - Σ(_ACCESS_DENY)
PRESENCE = If (Σ(_ABSENCE) > 0) 1 - Σ(_ABSENCE)
           Else Σ(_PRESENCE)
```

Proximity, collision and minimality are simple counter. Access and presence are more complex. Indeed a single _ACCESS_ALLOW is enough to cancel any number of _ACCESS_DENY. Identically, a single _ABSENCE cancels any number of _PRESENCE.

Geometry agents activity: ③ **Geometric transformations**

The last activity performed by geometry agents is to carry out a geometric transformation on the feature it represents. This is done on reception of requests from a coupled behaviour agent from other features. Geometric agents are in charge of modifying the geometric parameters defining the features (position, orientation and dimensions) to reflect the various geometric transforms supported by MADSfm. However, the computation of new parameters represents only a fraction of the work required. Indeed, after each geometric transformation, geometric agents have to re-evaluate their local beliefs database and propagate the changes to the rest of the agent community.

6.3.4.c Behaviour Agents

A behaviour agent is the brain of a design feature. Its only function is to optimise the welfare/fitness of the feature it inhabits. To this effect, a behaviour agent uses the belief list, its pre-defined desires, a set of predefined course of actions and an algorithm to choose the most adequate plan according to the current situation.

The behaviour agent is activated by a notification message from its coupled geometry agent. These notifications are sent to it by its geometric counter-part whenever a change occurred in the feature's beliefs. On notification, the behaviour agent starts a series of tests that compares its desires against the new beliefs. If mismatches are detected the decision engine is triggered in order to take actions and optimise the feature's fitness. The chosen

course of action can consist of requests to the coupled geometry agent to change the feature's geometry or requests to other agents.

The activity of a behaviour agent can be divided into three sub-tasks:

1. **Detecting Problems:**
   Whenever triggered by its geometric counter-part, the behaviour agent will compare its internal desires with the Beliefs kept by the Geometry Agent. Mismatch between corresponding desires and beliefs are flagged and later used in the problem-solving sub-task of the agent.

2. **Taking Decisions:**
   On finishing the detection phase, the behaviour agent checks its mismatch flags. If at least one flag is up the agent initiates its inference engine in an attempt to maximise its welfare (mismatch flags decrease the agent fitness!).

3. **Making it happen:**
   The behaviour agent is also in charge of putting its intentions into action. This is achieved by sending requests to its geometric counter-part or to other agents. If a planned action consists of a change in the feature's geometry, an internal request is sent to the geometry agent to change its internal data structures to reflect the new geometry. Requests can also be sent to other agents in the model, through KQML, when co-ordination is needed to maximise local fitness.

Detailed descriptions of these three sub-tasks are presented in the following sections.

Behaviour agents activity: ① D esire/Belief mismatch detection

Every time a change is made to a feature's belief, its geometry agent sends a triggering signal to its behaviour agent. This signal starts behavioural activity, which begins by detecting mismatches between feature's beliefs and desires. Simple comparison between the feature's pre-defined desires and the global beliefs generated during geometric analysis lead to the creation of mismatch flags.

Behaviour agents activity: ② Behaviour selection

Once existing mismatches have been flagged, the behaviour agent must select a suitable course of action from its plan database. Two separate mechanisms are involved in behaviour selection inside the system. Firstly, MADSfm supports a user-level behaviour selection through the Swarm GUI. This functionality allows the designer to select desired behaviours

for individual features inside a model. Throughout the life of a feature, the user has the possibility to choose alternate solving strategies for each validation criteria. This choice determines which plan will be applied when a mismatch is detected.

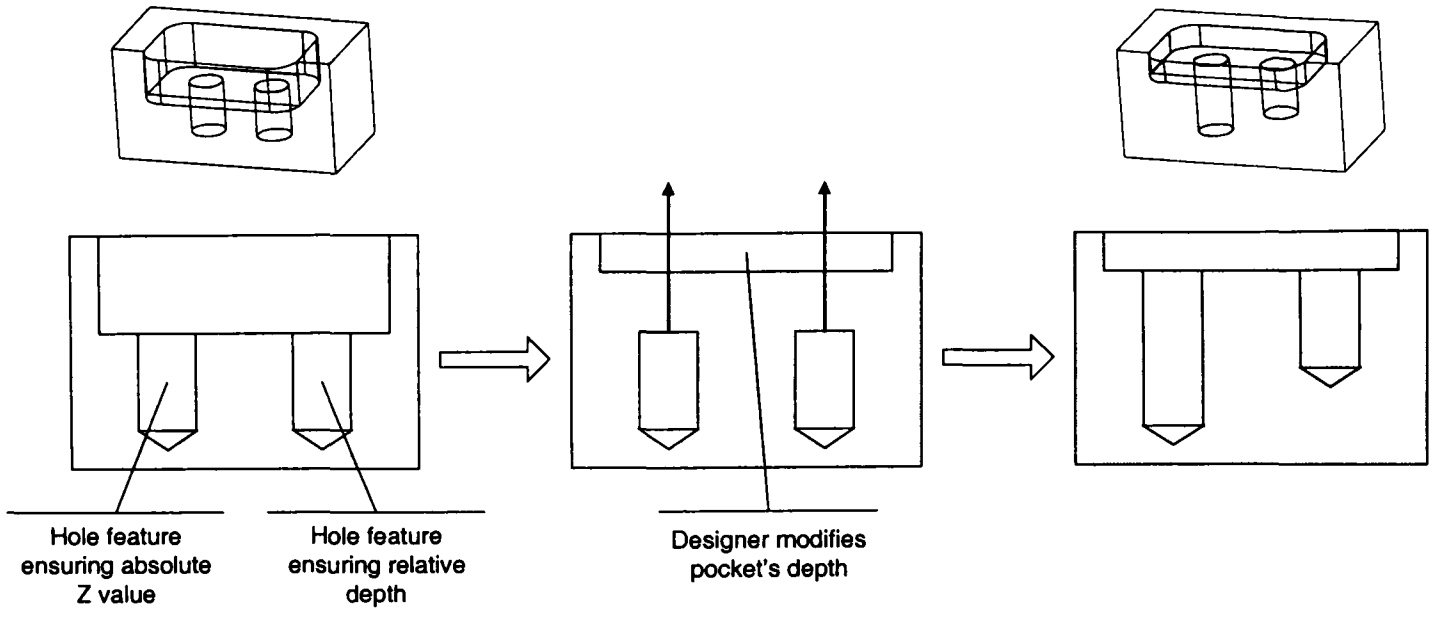


Figure 6-24: Alternate behaviours ensuring tool access

For example, in the case of ACCESS illustrated in Figure 6-24, the designer can choose between features that guarantee their relative depth and features that guarantee the absolute Z value of their bottom. Each manufacturability criteria possess an associated default solving behaviour and optional alternate behaviours. This second selection mechanism is also used to disable individual validation criteria of features by setting the desired behaviour as *inactive*.

Secondly, subsumption relationships (see section 4.3.5.b) exist between the different elementary behaviours of features, which define solving priorities for behaviour agents.

```
1 PRESENCE          ensures contribution to finished part
2 ACCESS            ensures machining can take place
3 PROXIMITY         takes process limitations into account
4 COLLISION         takes process limitations into account
5 MINIMALITY        optimises cutting operations
```

These relationships are used whenever several mismatch flags are up simultaneously to select which problem to solve first. This particular hierarchy can be justified from a machining point of view. Feature PRESENCE inside the finished part is paramount. Solving other problems before PRESENCE is useless since without PRESENCE, a feature can be eliminated from final machining. Next is ACCESS because without it, no actual machining can take place. PROXIMITY and COLLISION introduce the limitations inherent to the machining process. Their relative priority can not be easily justified, mostly because of the *special case* nature of collisions. Finally, for feature with no other problems, MINIMALITY introduces a degree of optimisation to the cutting operations required to obtain the finished part.

Selecting a behaviour is a two step operation. First the mismatch flags are collected and filtered to keep only the flags whose associated behaviour is not inactive. If several mismatch flag remain, the priority rules are applied to pick the user-selected behaviour for the most critical flag.

Behaviour agents activity: ③ **P r oblem solving behaviours**

The problem solving capabilities of MADSfm are simple and make full use of the various geometric elements generated by geometry agent during belief update. The five strategies implemented by MADSfm are described below and illustrated in Chapter 7 (see sections 7.4.1 to 7.4.4).

- 1. Presence behaviour:

  Inside MADSfm, positive features do no actively ensure their contribution to the finished part. It is felt that completely removing (through machining) the blank during design is too obvious a design mistake to warrant automatic solving. Ensuring presence of negative features inside the finished component is a more reasonable task. It is handled and always solved along the Z-axis. A search of potential access routes below (smaller Z co-ordinate) the feature is performed. If successful, the feature translates along Z to the closest one.

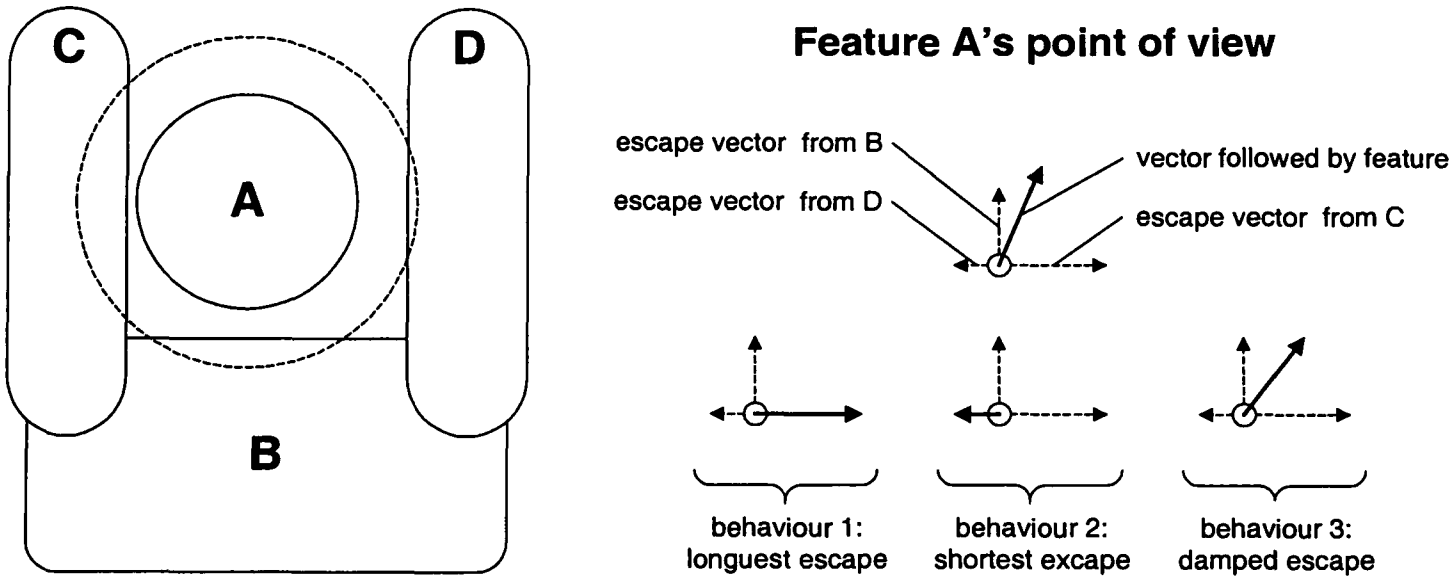


Figure 6-25: Alternative proximity avoidance behaviours

- 2. Proximity behaviour:

  Thin sections are dealt with within the XY plan. Basic "avoidance" strategies are used to determine how to eliminate proximity problems. During geometric analysis, a feature calculates the shortest distance between itself and others. An individual *escape vector* is also computed as being normal to the feature's profile

at the point of minimal distance[7]. These normal vectors are used to determine the direction and amount of XY translation necessary to ensure good clearance between features. A naïve strategy uses only the *escape vector* generated for the closest feature as shown in Figure 6-25 (behaviour 1). Although it eliminates one thin section, it does nothing to reduce potential side effect to the rest of the component. Another method is to use the shortest available escape vector to minimise geometric changes (see behaviour 2 in Figure 6-25). However, these two naïve approaches can only deal with one thin section at a time. A more refined method combines all generated escape vectors into a global vector before using it for translating the feature (see behaviour 3 in Figure 6-25). This effectively damps the feature's movement in tightly populated model, thereby reducing potential side effects. In particular, it prevents non-intersecting features to become intersecting as a result of proximity solving. In some cases, this damping can even prevent the feature from moving at all, in which case it relies on the other features to eliminate thin sections.

- 3. Access behaviour:

  Only negative features use access behaviours. Within MADSfm all access issues are solved along the Z-axis. All the implemented strategies for gaining tool access involve a search of the component for potential indirect access routes. A features (or union of features) is a potential candidate to grant tool access if it possesses the following properties: its XY profile surrounds the feature's profile, its bottom face lies higher (greater Z co-ordinate) than the feature's top face. If the search yields more than one candidate, the lowest (smallest Z co-ordinate) one is always elected to minimise model changes. If no potential candidates are found among negative features, no indirect access is available to the feature. In this case, direct access must be found to ensure accessibility. This is achieved by electing the top most positive feature currently denying access to the feature. A target Z value is extracted from the elected feature that represents where the feature's top face should lie to ensure accessibility.

  Once a candidate feature has been elected for providing tool access, the feature can modify its geometry to gain tool access. Two alternative strategies are available to the feature as shown in Figure 6-24. A feature preserve its relative depth by simply translating itself along Z. It can also preserve the absolute Z value of its bottom face by simultaneously increasing its depth and translating itself.

[7] This is possible because only convex feature profiles are currently allowed.

- 4. Minimality behaviour:

  Only vertical (along Z) minimality is currently supported in MADSfm. The minimality solving behaviours are therefore very similar to access behaviours. They involve a comparable search of the component for indirect and direct access routes. However, it looks for potential access routes located below (smaller Z value) its current position as opposed to higher in the case of access behaviours. Such accesses, if found, ensure feature minimality. Minimality also shares the alternative behaviour with access solving. A feature behaves consistently in this respect and uses the same technique to solve both access and minimality problems. This logical approach allows a feature to efficiently capture the designer's intent (see section 2.4.1). Indeed, a feature constrained as having a constant relative depth will behave consistently during both access and minimality solving.

- 5. Collision behaviour:

  There are no implemented behaviour that handles collision involving negative features. The only implemented collision behaviour is for collision between positive features. A search is performed in the XY plan to determine the shortest escape vector. The positive feature translates itself using the determined vector in order to eliminate the collision.

It can be noted that MADSfm favours geometric transformation along the Z-axis in most solving behaviours. Indeed, features currently only perform XY transformation to eliminate thin sections of the design. This preference can be justified by the fact that vertical transformation are easily reversed and have less potential side effects of the design than XY transformations. Although, such limitations are acceptable for a prototype system, it is obvious that more complex behaviours are needed for production CAD systems.

6.3.4.d Service Agents

Service agents are a great demonstration of the flexibility offered by multiagent systems. New agents can be added to existing systems that enhance existing functionality or create new capabilities with minimum effort. In fact, it is even possible to add new agents inside running systems and immediately benefit from their competence. Indeed, powerful ACL (such as KQML) provide dynamic registration mechanism that allow newly created agents to immediately take part in the community's activities.

Service agents provide optional high-level functionality that can be added to the system. In the case of MADSfm, their activity can add new functionality such as 3D display of components or agent's monitoring for livelock detection. They can also be used to increase system performance by providing some internal optimisation. A presentation of the four service agents implemented by MADSfm follows.

Service agents in MADSfm : ① **Display agent**

The display agent provides visual feedback to the designer by generating a 3D view of the designed component in the ACIS® 3D Toolkit. During its initialisation, the display agent creates a DDE (Dynamic Data Exchange) link with ACIS® that will be used during the entire design session. It also starts sending commands to the scheme interpreter to set up a viewing window and perform various initialisation operations. During normal operations, the display agent monitors any geometrical changes inside the model and translates them into Scheme commands. The generated commands are sent through the DDE link for ACIS to interpret and transform into a 3D representation of the component. The monitoring activity is simple. Because the display agent is a registered agent of the KQML facilitator, it receives all broadcast communication. It can extract all messages containing geometric data (using the ontology: geometry) and decide if new commands should be sent to ACIS®. Scheme entities inside ACIS® and features inside MADSfm are given an identical symbolic name in order to ease synchronisation of the two.

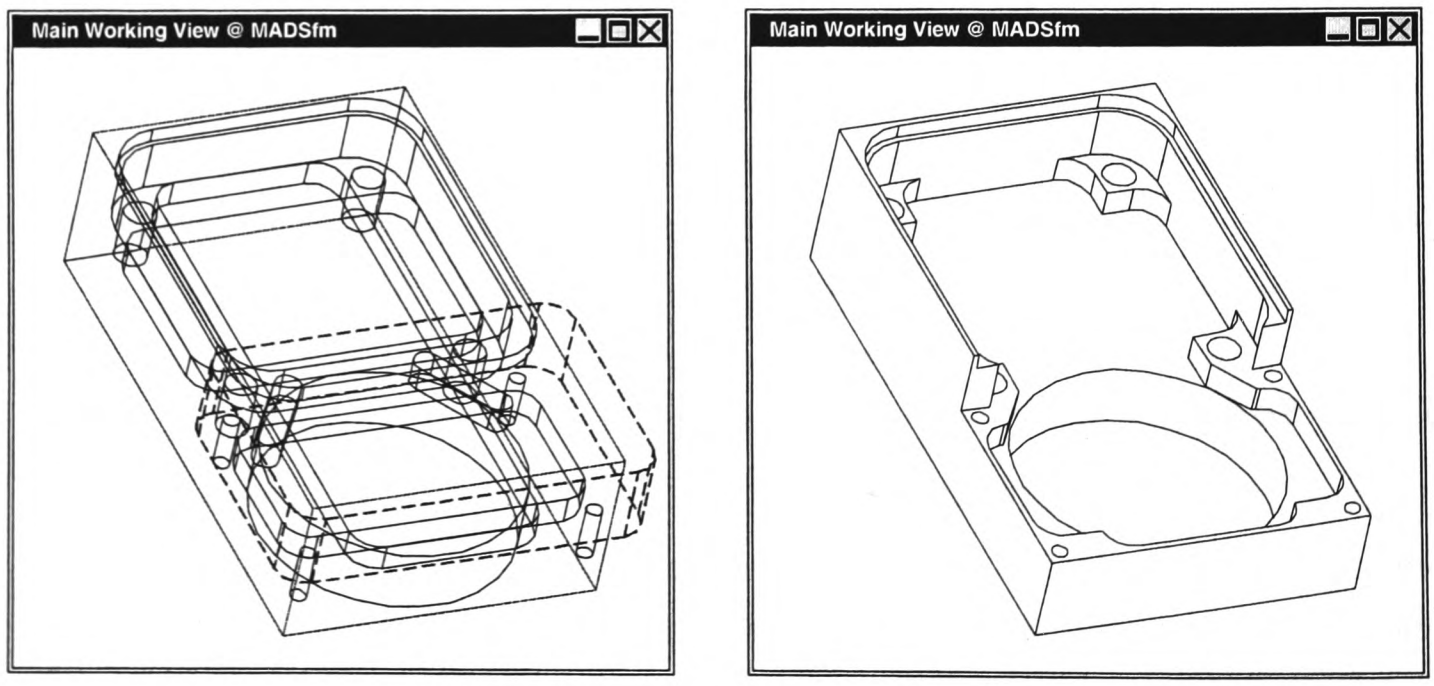


Figure 6-26: Examples of display agent's output

Special commands sent by the MADSfm's GUI are also processed by the display agent and illustrated in Figure 6-26. Visual selection of features inside the model is made possible. On clicking the "*select next*" or "*select previous*" buttons, the user actually sends a KQML request (through the user agent) to the display agent and the later changes the colour of the selected feature on screen. The display agent also supports a preview mode of finished parts, also accessed through a button on MADSfm GUI.

The ACIS® 3D Toolkit allows interactive manipulation of the generated 3D view. The user can use mouse and keyboard to zoom, pan and rotate the view. However, the DDE link used by the display agent is unidirectional and it provides no feedback from ACIS® to MADSfm.

Service agents in MADSfm : ② Constraint manager

Mechanical design often involves the specification geometric constraints between entities. Feature-based design is no exception. Feature themselves hold a number of low-level geometric constraints in their specification. For example, slot specifications usually contain a constraint for ensuring co-planar faces on each side. Because of this, feature-based design does not require designers to specify large numbers of constraints. Yet, it is still desirable to be able to express constraints between features. The constraint manager agent offers this additional functionality to MADSfm.

The internal operation of the constraint manager is basic. It performs automatic constraint propagation throughout the system with no checking for possible conflicts. The manager maintains a list of existing atomic constraints. As a registered agent, it receives all broadcast messages and uses them to monitor changes inside the model. When a modification is detected, it searches the constraint list for constraints involving the modified feature. When an appropriate constraint is found, the manager simply applies the constraint relaxation rule contained in the constraint definition and resumes the search in the list.

The constraint manager inside the system is a late addition to the MAS. It is implemented to demonstrate the feasibility of mixing agent-based modelling with conventional constraint based modelling. Because it was never intended to be a functional constraint manager, it extremely limited and only supports concentricity constraints. The inclusion of a truly functional constraint solver such as the SkyBlue solver **[188]** is seen as a feasible task.

Service agents in MADSfm : ③ **Space partitioner**

The space partitioner is an important addition to MADSfm because it limits features to a local point of view inside the system thereby reducing the overall communication load. It uses the concept of octree **[189, 190, 191]** to determine feature locality inside a part.

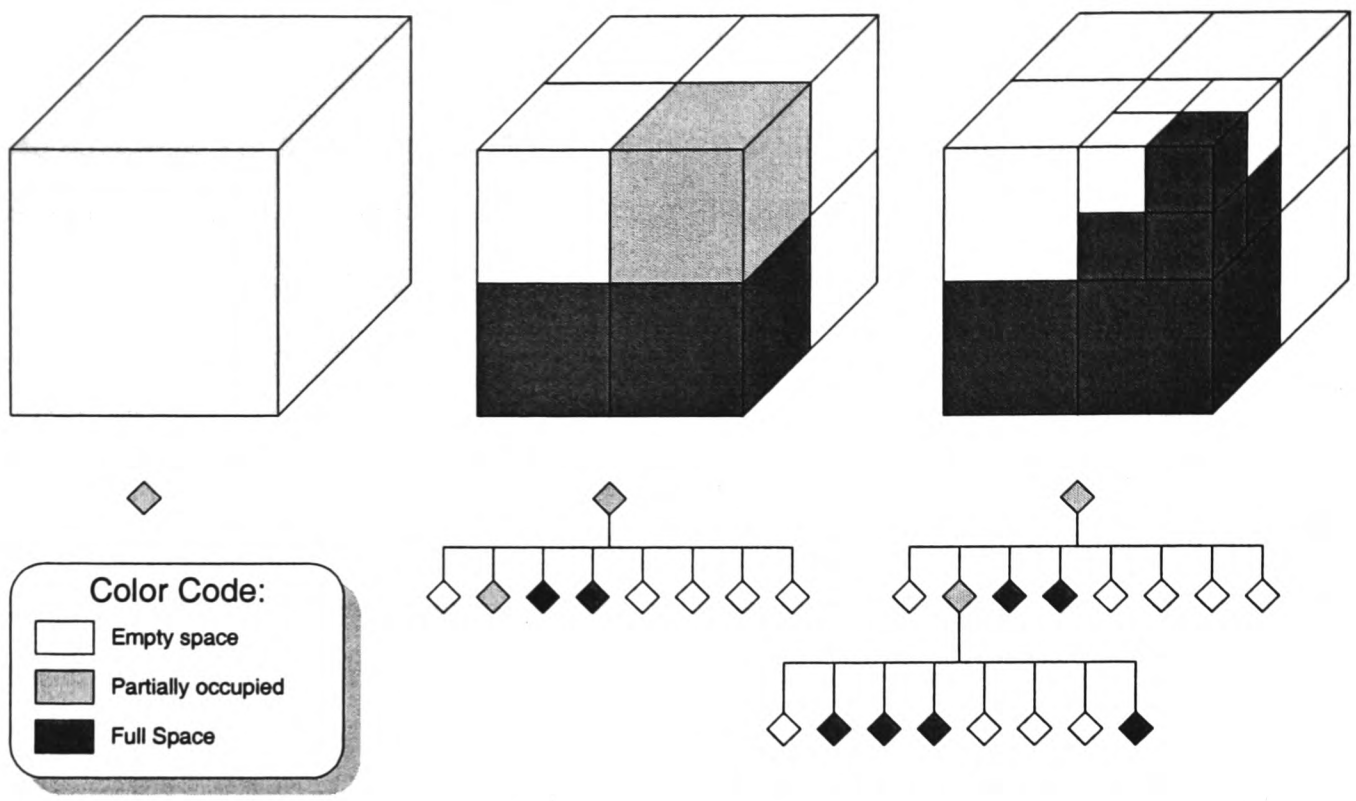


Figure 6-27: Octree decomposition

An octree describes objects by recursively partitioning 3D space into marked sub-spaces stored in a tree-like data structure. It requires the definition of a initial universe, which marks the area to be encoded, before decomposition is performed. The decomposition process consists in evaluating units of space (starting with the initial universe) against the object. If a unit of space is totally located inside the object it is marked as *full* (or black). If it is totally outside the object it is marked as *empty* (or white). If the unit of space partially intersects with the object it is marked as *partial* (or gray), then the partial unit is split into 8 sub-units of space and the process is recursively carried out on each sub-unit. The process stops when partial octants exist or when a fixed level of recursion is reached as shown in Figure 6-27. This level of recursion defines the resolution (or accuracy) of the octree.

Octrees offer several interesting qualities for tracking locality inside MADSfm:

- They are based on the concept of space partitioning
- They are efficient for detecting intersection between objects
- Their hierarchical nature permits operations at different resolutions

- They are computationally inexpensive to build and maintain

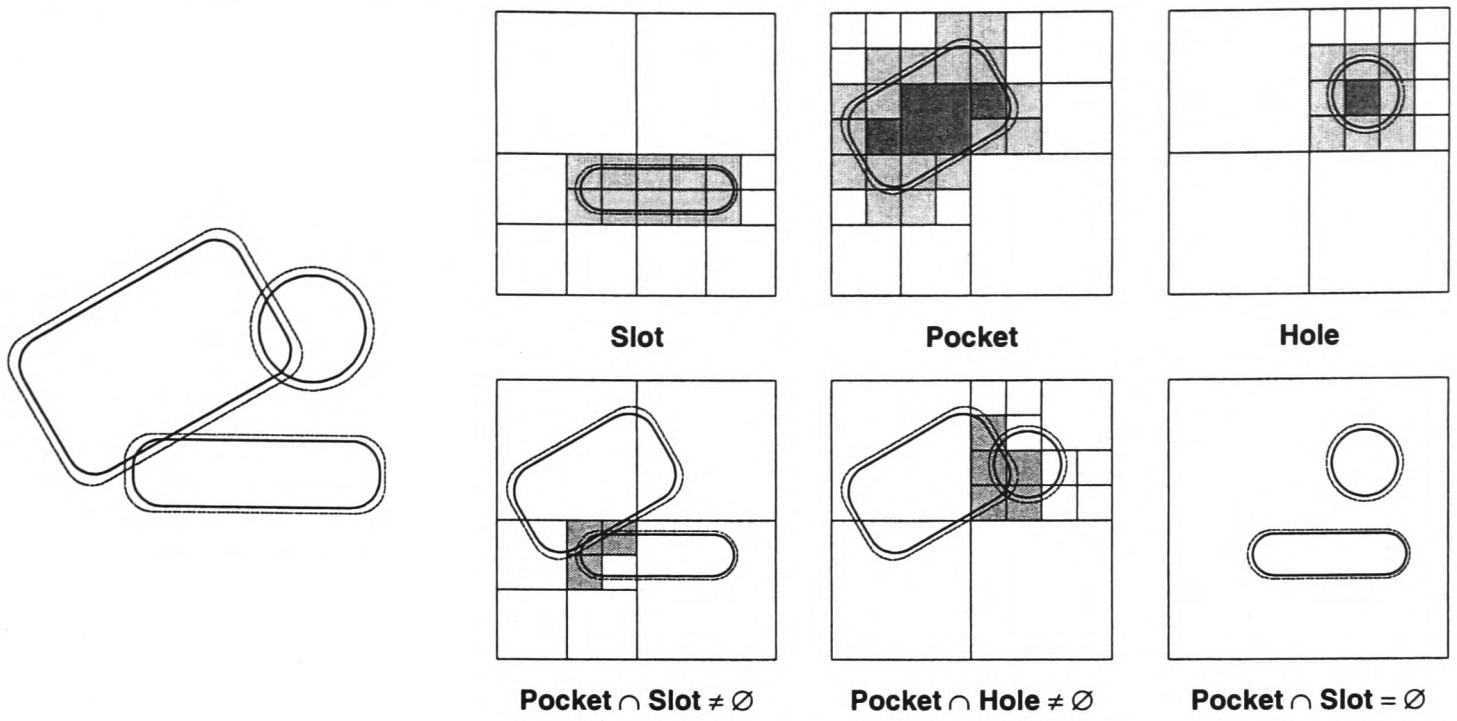


Figure 6-28: Determining feature locality with quadtrees

The space partitioner builds an octree representation of each feature in the system and uses them to determine locality relationships between features. Figure 6-28 illustrates the quadtree (2D equivalent of octree) encoding of three features and the determination of their locality relationships. Although it only shows 2D, the principle remains identical for octrees. Each feature is grown by a pre-defined amount before being encoded in separate trees. This growing of features is necessary to include proximity inside the concept of locality. Once each feature tree has been created, locality is simply determined by intersecting each pair and testing the resulting tree. Empty results reveal non-local feature, while non-empty mark locality relationships.

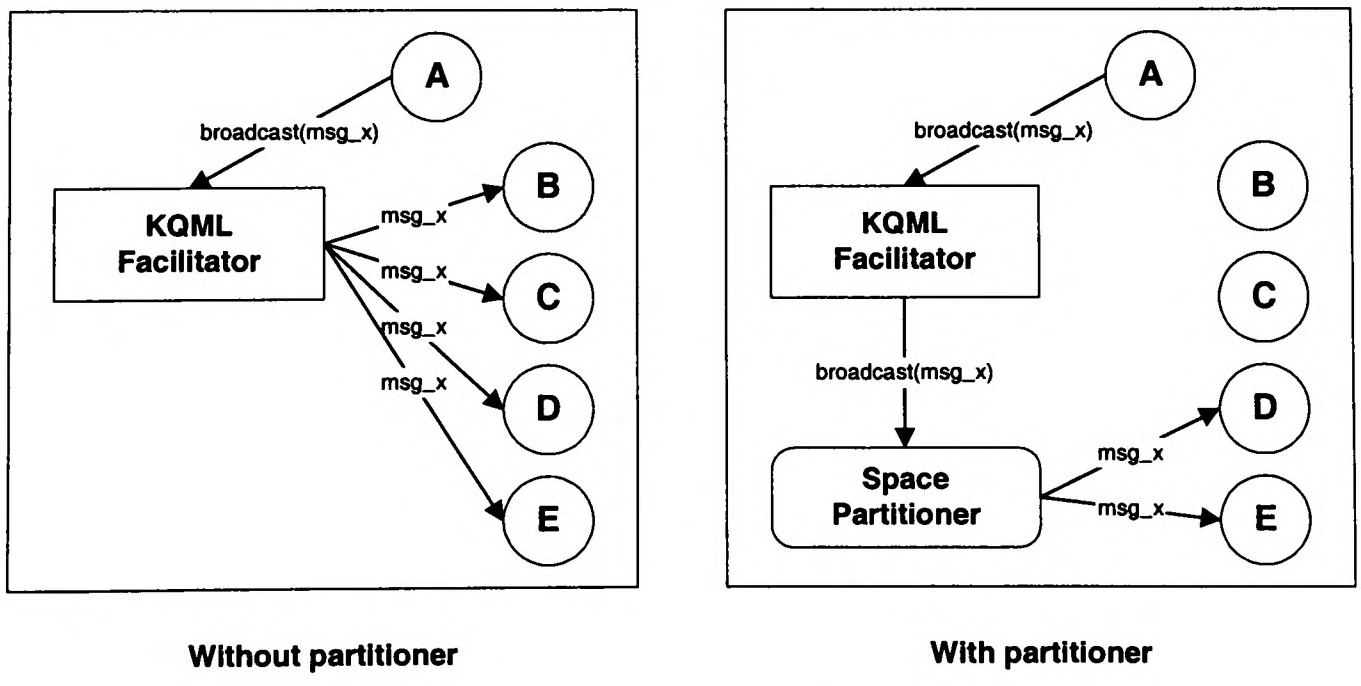


Figure 6-29: Broadcast mechanisms in MADSfm

When added to the system, the space partitioner registers with the KQML facilitator and start offering its communication partitioning services to it. Instead of dispatching broadcast messages by itself the facilitator delegates that task to the partitioner. Broadcast KQML messages sent to the facilitator are forwarded to the partitioner, which determine the features that should receive it based on their locality to the sender (see Figure 6-29). For simple designs with a small number of features, the overhead represented by the addition of space partition is not justified. However, as the number of feature increases, the gain in terms of communication load and agent processing becomes significant. Indeed, the space partitioner must perform complex geometrical computations before dispatching a message. But in large models this can prevent all features having to process the incoming geometry themselves, thereby making the addition worthwhile.

Service agents in MADSfm : ④ **Activity monitor**

The potential for livelocks inside multiagent systems has previously been discussed in 6.2.1.a. Livelocks can be damaging in a design system like MADSfm because it can induce unnecessary activity, which may lead to loss of a partial solution inside a part. They are characterised by endless event loops between two or more autonomous agents. The activity monitor is added to MADSfm in an attempt to detect livelocks and offer a recovery mechanism in case where the design gets badly damaged by it.

MADSfm is based on Swarm and uses self-scheduling agents, which share a common activity schedule. This architecture has the advantage of easing the monitoring of activity inside the agent community. Indeed, all activity must go through the common schedule, which become the ideal place to supervise global activity. The activity monitor continuously watches the common schedule to assess the status of the MAS.

An empty schedule is the sign of inactive feature agents, which in turn reflects a stable design configuration. It does not necessarily mean that it is an optimised or even a fully manufacturable configuration. It is characteristic of a stable design in which agents have either fulfilled their goals or reached the limit or their knowledge. When such a stable situation is detected, the activity monitor stores a *snapshot* of the design that represents the current stable state.

Livelocks are characterised by uninterrupted agent activity, which translate into a schedule that never becomes empty. The activity monitor can suspect livelocks when its last snapshot of the system becomes too old. However deciding on an acceptable time limit before

declaring a livelocked state is a difficult matter. A low limit might wrongly detect livelocks in very active systems containing large number of features. A high limit can keep the system locked too long thereby reducing responsiveness and availability. Empirical limit values are used in MADSfm that are also dynamically adapted to the number of features living in the model. When a livelock situation is detected, the activity monitor has two options. It can disable all behaviours on livelocked features in order to end the feedback loop or it can revert the entire design to the last stored snapshot. The first solution is preferable when no major damage was done to the design, while reverting to the previous stable state is useful when things go really wrong.

The activity monitor uses simple rules to determine whether the MAS is healthy or not. Observation of the central activity schedule provides an efficient way of determining MADSfm's state. No expensive computation is needed and simple timing of uninterrupted activity is enough to recognised livelocked agents. The activity monitor adds design recovery through its capability to save and restore previous stable designs.

6.3.4.e Change propagation

In the newly created active models, feature agents can modify their geometry autonomously to ensure manufacturability. In this context, the propagation of changes throughout the model is critical to guarantee proper operation. The activity of the system is maintained by an autonomous flow of geometric data between agents in the system.

Peer to peer communication inside the agent community is the root of all activity within the designed component. Two basic mechanisms are implemented in MADSfm that are responsible for keeping the agent community alive and responsive at all time.

- Whenever its internal geometry is modified (including creation and destruction), by the user or by itself, a feature agent automatically broadcasts a notification message to all other agents in the system. This *reflex* has the effect of propagating any geometrical changes to the entire system, thereby ensuring that all agents have up-to-date information at all time.

- On reception of such a notification message, a feature agent systematically analyses the new geometry against itself and applies its embedded knowledge to detect and solve potential manufacturing problems. If a problem is detected during analysis that

requires the feature to modify its own geometry, it will in turns broadcast this change to the rest of the agent community.

This basic scheme proved efficient. MADSfm is able to propagate any geometric changes that occur as a result of both user and agent activity. Propagation of changes by message broadcasting allows the system to responds immediately to any occurring changes. Moreover, it allows a dormant system (stable model) to automatically awaken in response to geometrical modifications.

6.3.4.f Agent learning

It has been explained in 5.3.4 that feature agents need to learn their environment before they can perform their activity. A newly introduced feature advertises its presence to the rest of the model by broadcasting its geometry. Other features in the system reply to this broadcast by returning their geometric properties. With this simple mechanism, two important tasks are accomplished. Newly added features are always acknowledged by existing features and environmental data is provided autonomously to new features. However, communication latency and global system activity introduce a delay between the introduction of a feature and its full awareness of its environment. This delay has proven to be damaging for the model, because new features tend to take uninformed decisions when applying their behaviours. This is especially true when loading a part definition from file as described in 7.3.1. To eliminate this drawback, MADSfm provides a "fast learning" mechanism for newly created feature agents.

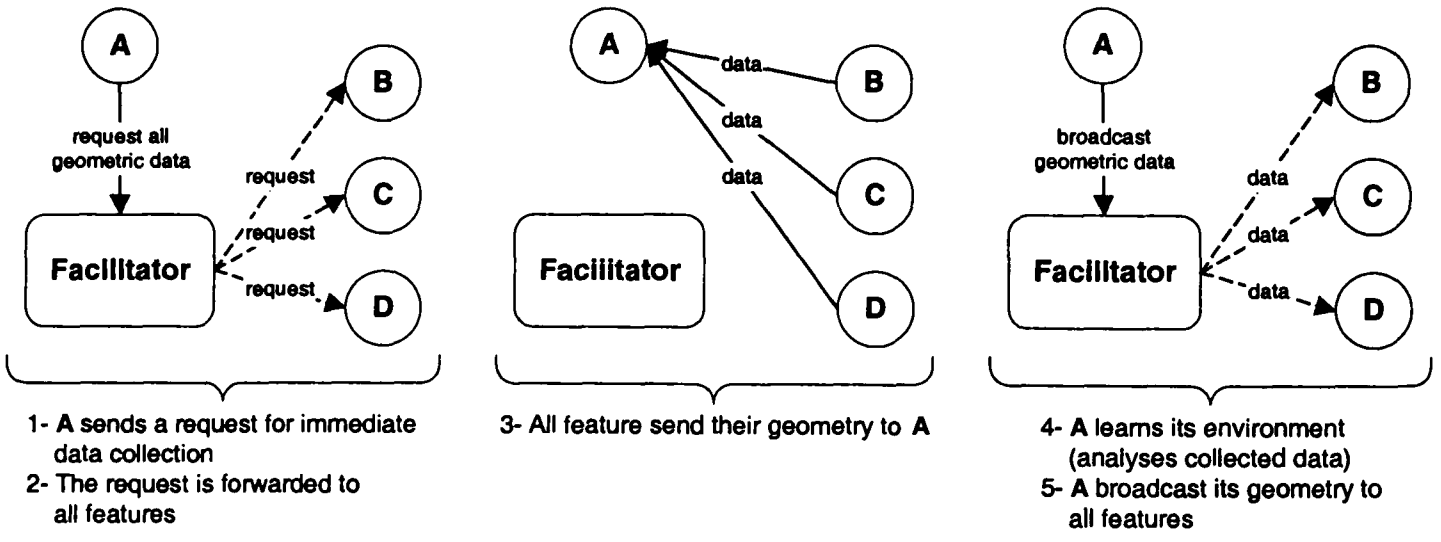


Figure 6-30: Fast learning during feature creation

Figure 6-30 illustrates how it operates. New features send a request for immediate transmission of geometric data to the system's facilitator (see section 6.3.3.d). This special

request is forwarded to all features, and is acted upon immediately. Every feature in the system responds by sending their geometric data to the new feature. The later analyses this data, and generates a new set of beliefs before starting its activity. The new feature can thereafter broadcast its geometry to make itself known to other features.

By using this fast learning scheme, the restoration of parts from files performs without undesired activity due to feature without local knowledge of their environment. Indeed, by the time all features are created and the global activity started, all agents fully possess the local data they required to operate.

6.3.4.g Dynamic behaviour selection

Features agents perform their activity based on manufacturability criteria, which express some knowledge of the production process. A list of alternative solving routines (or behaviours) corresponds to each of these criteria. These behaviours usually implement different solving strategies for particular manufacturability criteria. This mechanism introduces a high degree of flexibility to the system. The user can use behaviour selection to achieve different objectives.

- One of the major challenges that modern CAD packages must address, is the successful capture of the designer's intent during the design phase. It was explained in 2.4.1 that failure to capture the user's intent properly could result in undesired results (see Figure 2-8) during editing of the model. The agent-based approach presented in this thesis introduces an original way of capturing this intent through the selection of alternative feature behaviour during design. At any time during a design session, the designer can select his preferred solving routine for individual manufacturability criteria of individual features. Alternative routines generate different feature responses to detected problems as explained in 6.3.4.c. Consider a through hole added to an existing design. If it is the intention of the designer to preserve this hole as through, he/she would create a hole deep enough to go through the blank at creation time and select behaviours (access and minimality) that preserve the absolute depth of the feature. This way, the through hole is preserved even when its representing agent performs self-correction concerning its accessibility.

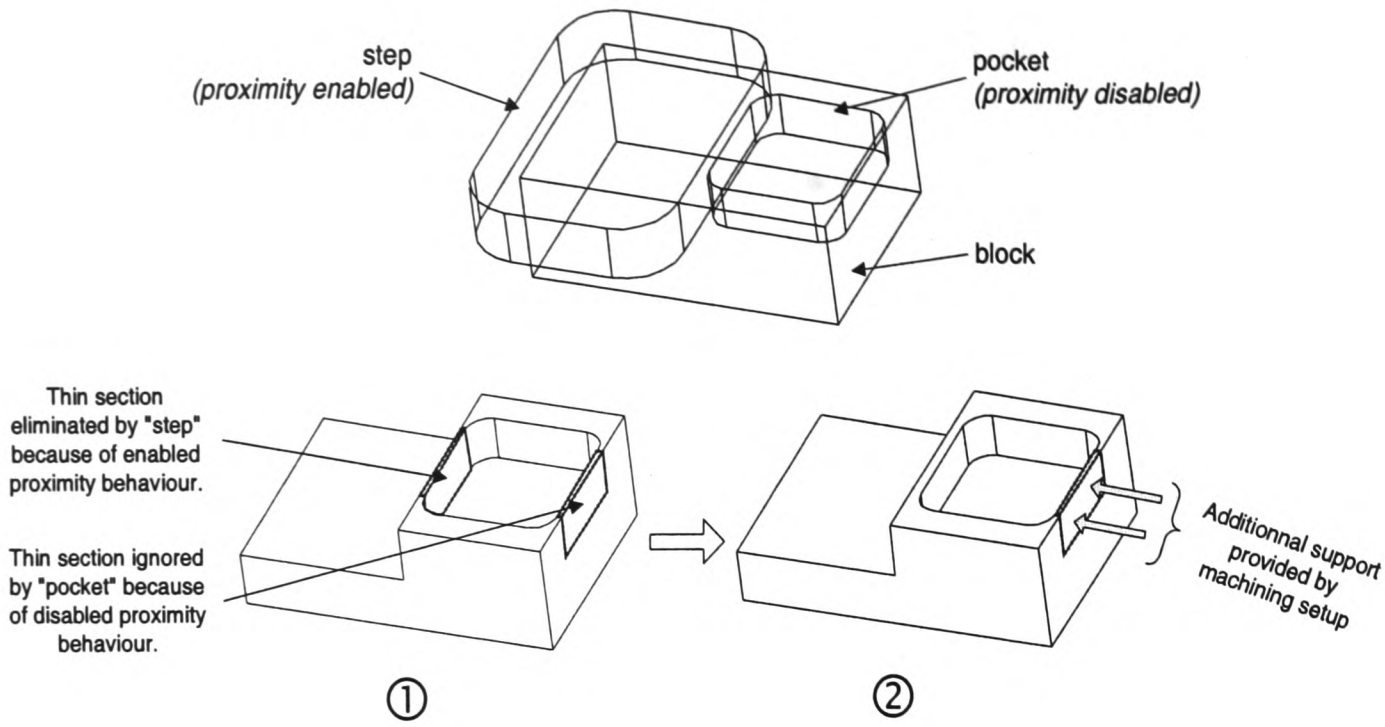


Figure 6-31: Using behaviour selection to ignore *desired* thin sections

- As an alternative to choosing an actual behaviour, the designer might choose to select the *empty* behaviour, which disables the solving knowledge concerning an individual validation criterion. This is a useful feature of the system because it allows the designer to override the feature's decision in problematic sections of the design. Notably, disabling behavioural responses does not prevent the geometric analysis activity of features. This is important as it allows part of a design to be frozen in its current state without crippling the detection and solving capabilities of the rest. Consider the example illustrated in Figure 6-31. A block's top face needs to be pocketed with the creation of potentially dangerous thin sections at the edge of the block. The default behaviour of the pocket is to eliminate this thin wall through self-correction. However, the designer might know that additional support would be available during machining to prevent flexion or rupture of this particular wall (see Figure 6-31②). In such a case, the designer is able to disable the proximity behaviour of the pocket to impose the design intent over the feature's goals. It should be noted that a thin wall between the pocket and the step could still be autonomously handled by the system. Moreover, because the pocket does not stop its analysis activity, it is able to report on the number of existing thin section to the user. This information can be used by the designer to determine if an unforeseen thin section exists before committing the design to process planning.

6.4 Conclusion

A number of crucial issues concerning the development of multiagent systems were presented at the beginning of this chapter. Concepts such as communication load, concurrency, reactive agency, BDI motivation, deadlocks and livelocks were discussed through a presentation of preliminary implementation work carried out during this Ph.D. The final test-bed for the feature as agent paradigm occupied the remainder of the chapter.

MADSfm is a MAS based on the Swarm libraries. It provides an interactive 2½D-modelling environment for the design of mechanical components destined to be machined on 3-axis milling-machines. The global architecture and operation of the system was presented in details. An active MADSfm design session contains a number of autonomous agents working together on behalf of the designers. Feature agents hold the geometric data forming a component. They divide their activity between geometric analysis and behavioural actions. This duality is at the core of the features agents, which are composed of a geometry agent and a behaviour agent. Geometry agents perform geometric analysis and generate beliefs about a feature's environment. Behaviour agents use these beliefs to apply internal skills that solve common design problems, which in turn lead to better manufacturability. Service agents support the MAS by offering high-level capabilities such as 3D display, communication optimisation or geometric constraints management.

This chapter provides in depth explanations of the architecture used to create feature-based agent-driven design system. It reveals the complexity involved in validating machining features and also offers a glimpse of how agent-based software engineering can tackle such problems.

Chapter 7

Results and Analysis

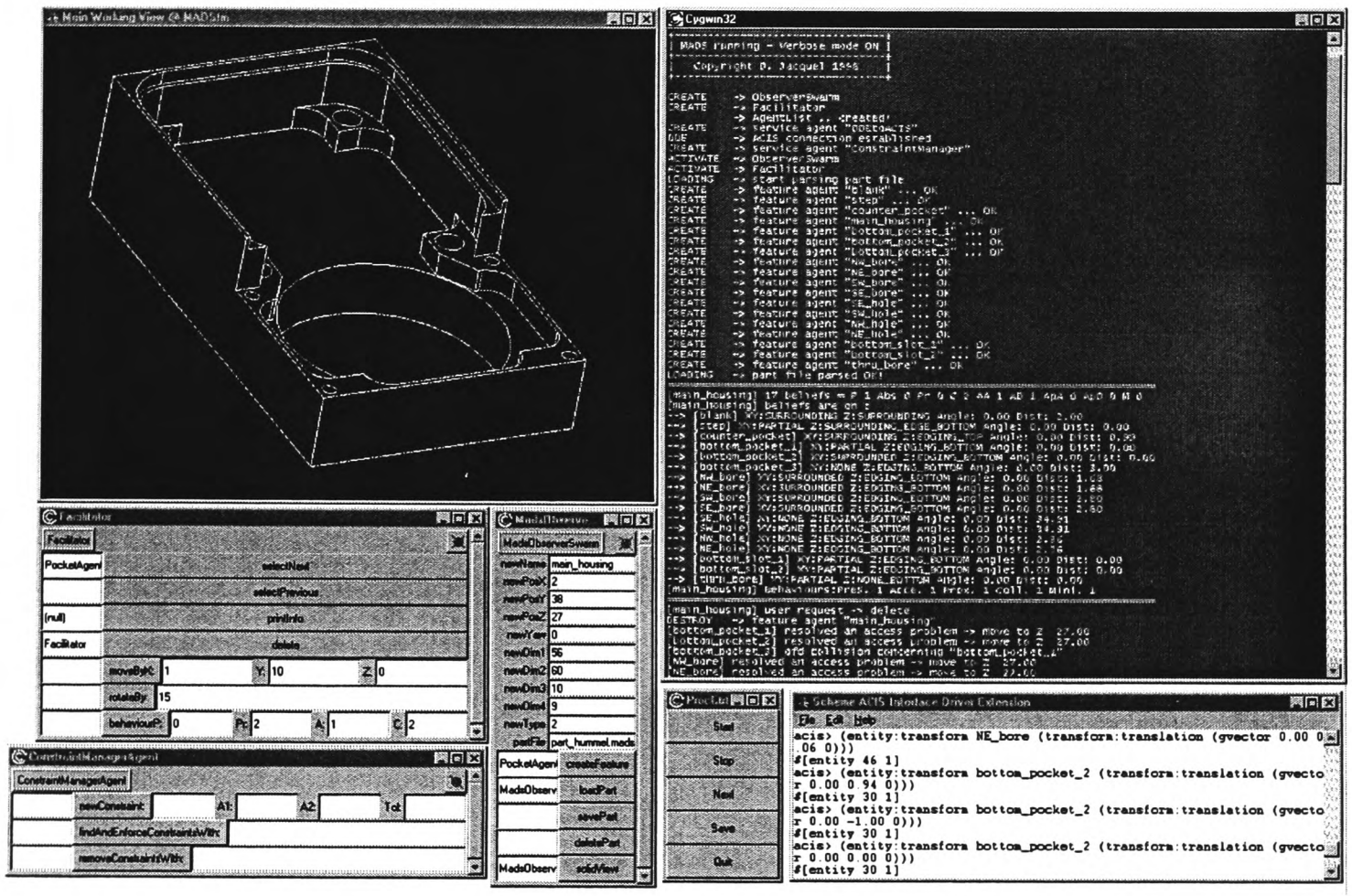


Figure 7-1: MADSfm in action

7.1 Introduction

Chapter 5 and Chapter 6 have discussed the general concept of autonomous feature-agents and a particular implementation of it. This chapter attempts to present and analyse the resulting system with respect to the different points put forward in Chapter 5. It principally aims to demonstrate the novel qualities of using MAS for feature-based design as well as

pointing out the limitations of the MADSfm (MultiAgent Design System for manufacturability) prototype implementation and illustrated in Figure 7-1.

7.2 Dynamic model

A major contribution of agency in the context of feature-based design is the creation of a dynamic product model replacing a conventionally passive data structure. Indeed a community of autonomous agents now represents the product being designed. Agents provide the user with their knowledge and perform tasks on his/her behalf. Because of this dynamic model, the global behaviour of the agent-driven system is radically different from a conventional CAD package.

7.2.1 Problem detection

Autonomous manufacturability analysis of designs is the first obvious advantage of using feature-agents. Features automatically detect potential problem concerning their manufacturability by autonomously testing themselves against other features in their environment. Various interesting issues relating to the problem detection activity are now discussed and illustrated using examples tested with MADSfm.

7.2.1.a Principle of locality

It has been previously seen that the principle of locality is crucial to agency (see section 4.2.2). Geometric analysis and problem detection inside MADSfm are carried out locally by each feature-agent. This locality in geometric analysis means two things. First, features need only know about their immediate surroundings in order to function. Second, features only detect manufacturing problems from their own point of view.

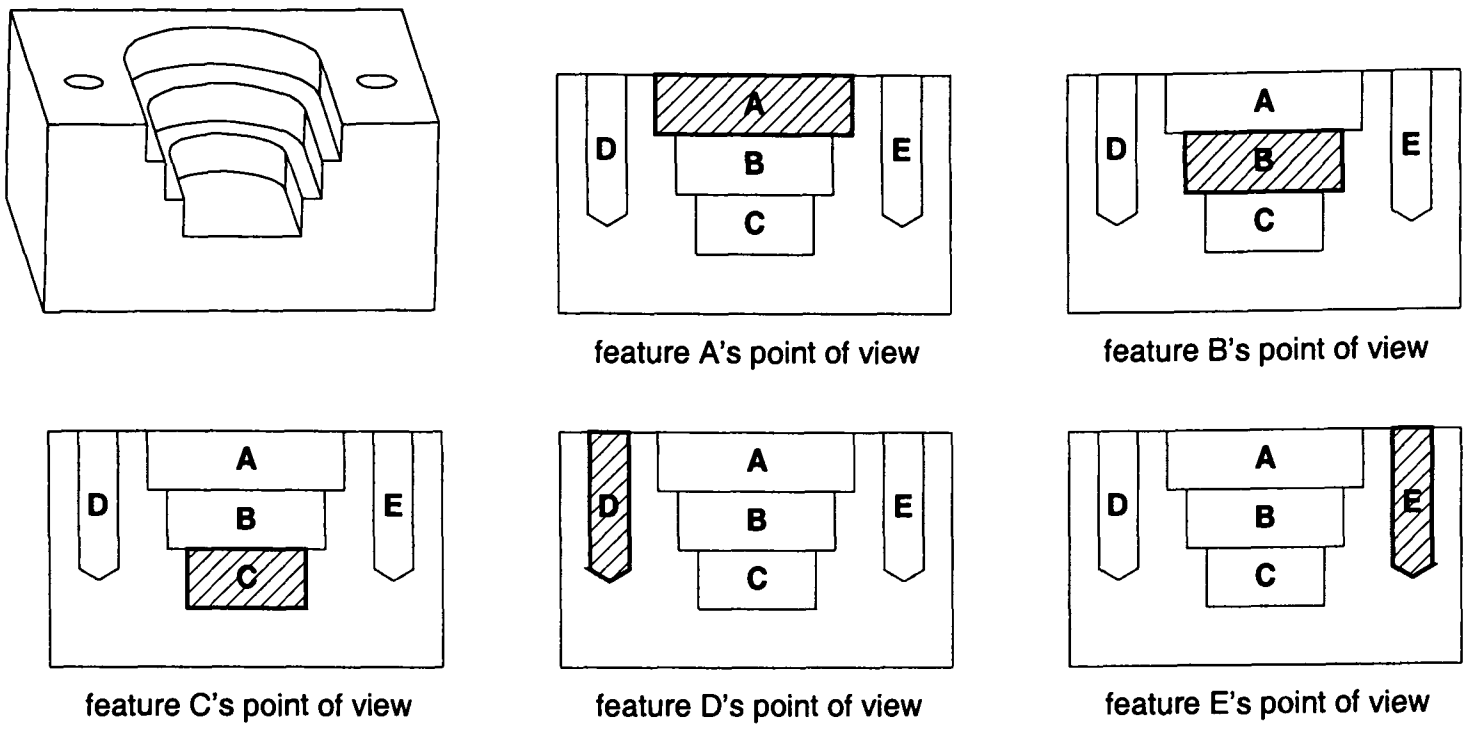


Figure 7-2: Locality of knowledge for feature agents

The principle of locality applicable to feature agents makes geometric partitioning and clustering possible within the multiagent model. Figure 7-2 illustrates how feature agents take advantage of the principle of geometric locality in order to minimise their workload and focus their attention on neighbouring features. The point of views of five negative features contained within the example part are shown individually.

The following listings (Listing 7-1 to Listing 7-6) present the debug output that can be obtained from feature agent inside the swarm console of MADSfm (see section 6.3.3). Individual features making the model in Figure 7-2 can be queried concerning their current beliefs. The formatting and symbols used in this output are as follows.

```
For global beliefs:     P       means PRESENCE
                        Ab      means ABSENCE
                        Abp     means PARTIAL_ABSENCE
                        Pr      means PROXIMITY
                        C       means COLLISION
                        AA      means ACCESS_ALLOW
                        AD      means ACCESS_DENY
                        AA      means PARTIAL_ACCESS_ALLOW
                        AD      means PARTIAL_ACCESS_DENY
                        M       means MINIMALITY
                        Mp      means PARTIAL_MINIMALITY

For individual beliefs: XY      intersection type in XY plane
                        Z       intersection type along Z
                        Angle   smallest intersection angle
                        Dist    smallest separating distance

    Manufacturability beliefs held as true are displayed, others are not.
    When no XY intersection exist, no angle exists and n/a is displayed.
```

```
[A] global beliefs = P 1 Ab 0 Abp 0 Pr 0 C 0 AA 0 AD 0 ApA 0 ApD 0 M 0 Mp 0
[A] 4 individual beliefs concern :
--> [block] XY:PARTIAL Z:SURROUNDING_EDGE_TOP Angle: 90.00 Dist: 5.00
  _PRESENCE
--> [B] XY:SURROUNDED Z:EDGING_BOTTOM Angle: n/a  Dist: 1.98
--> [E] XY:NONE Z:SURROUNDING_EDGE_TOP Angle: n/a  Dist: 3.00
--> [D] XY:NONE Z:SURROUNDING_EDGE_TOP Angle: n/a  Dist: 3.00
```

Listing 7-1: console output of feature A's beliefs

```
[B] global beliefs = P 1 Ab 0 Abp 0 Pr 0 C 0 AA 1 AD 0 ApA 0 ApD 1 M 0 Mp 0
[B] 3 individual beliefs concern :
--> [A] XY:SURROUNDING Z:EDGING_TOP Angle: n/a  Dist: 1.98
  _ACCESS_ALLOW
--> [C] XY:SURROUNDED Z:EDGING_BOTTOM Angle: n/a  Dist: 1.97
--> [block] XY:PARTIAL Z:SURROUNDING Angle: 90.00 Dist: 11.00
  _PRESENCE _ACCESS_P_DENY
```

Listing 7-2: console output of feature B's beliefs

```
[C] global beliefs = P 1 Ab 0 Abp 0 Pr 0 C 0 AA 1 AD 0 ApA 0 ApD 1 M 0 Mp 0
[C] 2 individual beliefs concern :
--> [B] XY:SURROUNDING Z:EDGING_TOP Angle: n/a  Dist: 1.97
  _ACCESS_ALLOW
--> [block] XY:PARTIAL Z:SURROUNDING Angle: 90.00 Dist: 14.00
  _PRESENCE _ACCESS_P_DENY
```

Listing 7-3: console output of feature C's beliefs

```
[D] global beliefs = P 1 Ab 0 Abp 0 Pr 0 C 0 AA 0 AD 0 ApA 0 ApD 0 M 0 Mp 0
[D] 2 individual beliefs concern :
--> [A] XY:NONE Z:SURROUNDED_EDGE_TOP Angle: n/a  Dist: 3.00
--> [block] XY:SURROUNDING Z:SURROUNDING_EDGE_TOP Angle: n/a  Dist: 3.00
  _PRESENCE
```

Listing 7-4: console output of feature D's beliefs

```
[E] global beliefs = P 1 Ab 0 Abp 0 Pr 0 C 0 AA 0 AD 0 ApA 0 ApD 0 M 0 Mp 0
[E] 2 individual beliefs concern :
--> [A] XY:NONE Z:SURROUNDED_EDGE_TOP Angle: n/a  Dist: 3.00
--> [block] XY:SURROUNDING Z:SURROUNDING_EDGE_TOP Angle: n/a  Dist: 3.00
  _PRESENCE
```

Listing 7-5: console output of feature E's beliefs

```
[block] global beliefs = P 5 Ab 0 Abp 0 Pr 1 C 0 AA 0 AD 0 ApA 0 ApD 0 M 0
[block] 5 individual beliefs concern :
--> [A] XY:PARTIAL Z:SURROUNDED_EDGE_TOP Angle: 90.00 Dist: 3.13
  _PRESENCE
--> [B] XY:PARTIAL Z:SURROUNDED Angle: 90.00 Dist: 2.50
  _PRESENCE
--> [C] XY:PARTIAL Z:SURROUNDED Angle: 90.00 Dist: 0.00
  _PRESENCE _PROXIMITY
--> [E] XY:SURROUNDED Z:SURROUNDED_EDGE_TOP Angle: n/a  Dist: 3.00
  _PRESENCE
--> [D] XY:SURROUNDED Z:SURROUNDED_EDGE_TOP Angle: n/a  Dist: 3.00
  _PRESENCE
```

Listing 7-6: console output of feature Block's beliefs

Two cases occurring in the model shown in Figure 7-2 deserve closer attention.

- Firstly, consider the situation of feature *C* that sits at the bottom of two nested features (*A* and *B*). It only considers the features *Block* and *B* to perform its local activity. Listing 7-3 shows the beliefs generated by *C* concerning these two local features. Block provides PRESENCE but partially denies tool ACCESS. *B* grants full ACCESS to *C*. Therefore *C* fulfils all its desires in this current configuration. The important fact to notice is that *C* is not interested in ensuring global accessibility. Instead, it only ensures its local accessibility through *B* and relies on the later to ensure its own ACCESS.

- Secondly, features *D* and *E* demonstrate XY locality within the model. Inspecting the beliefs of both features (Listing 7-4 and Listing 7-5), it emerges that they only take *Block* and *A* in their local activity but not each other. Indeed, the proximity of two features in the XY plane determines the locality of features that overlap along the Z-axis. Therefore, while *D* and *E* need to consider *A* for potential proximity issues, they can safely ignore each other.

It can be noted that in the case of a model containing a single positive feature, the later will necessarily have beliefs concerning every negative feature in the model. This is because the PRESENCE desire and behaviours of negative features ensures that geometric interaction exist between them and the blank.

7.2.1.b Problems involving more than 2 features

All geometric analysis and belief generation inside MADSfm is done locally by agents. Each feature maintains a local database of beliefs, which represent a vision of the

environment and is used to determine manufacturability status. An obvious drawback of this approach is that any problem involving more than two features must be treated as special cases and might be missed out.

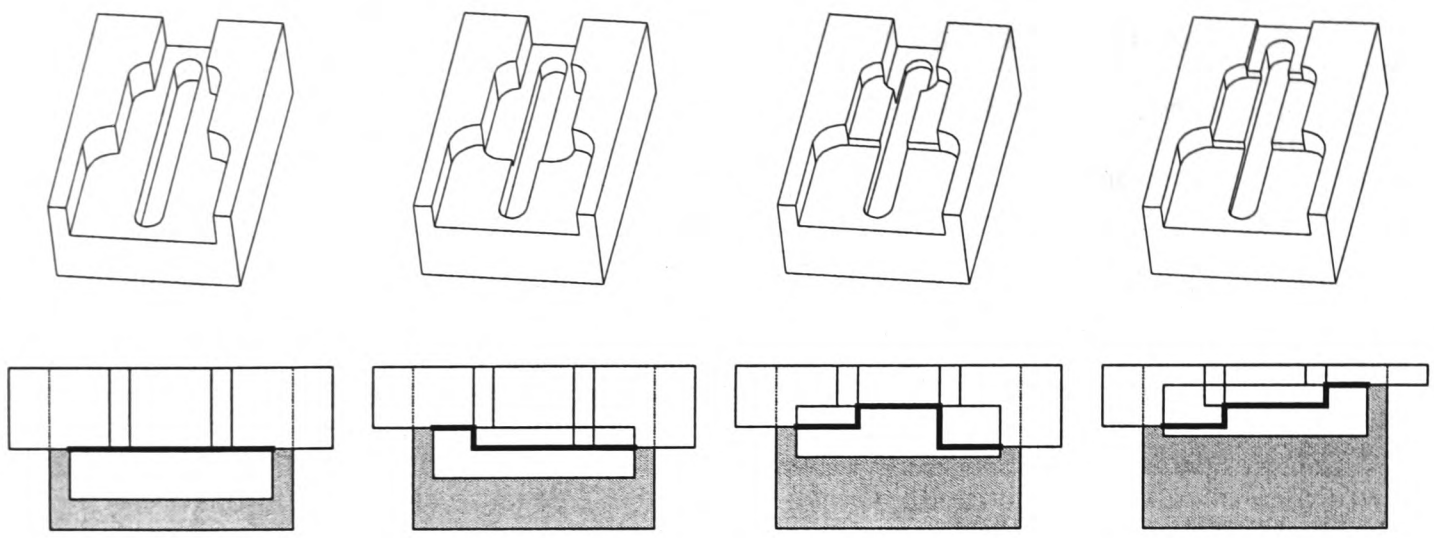


Figure 7-3: Partial access examples

A possible mechanism to handle partial contributions of features for tool access and minimality has been presented in section 6.3.4. Figure 7-3 illustrates different partial access configurations for a slot positioned at the bottom of three partially overlapping pockets. Tool access to the slot is granted through the combination of the partial access provided by each of the three pockets. Only checking for possible access through a single other feature would result in the bottom slot wrongly detecting access problems in all cases shown in Figure 7-3. The access detection algorithm used by MADSfm proves able to handle such partial accesses successfully. This success results from the fact that partial contributions are not considered individually. Instead, various combinations of partially contributing features are generated and tested to identify potential access provided through several features.

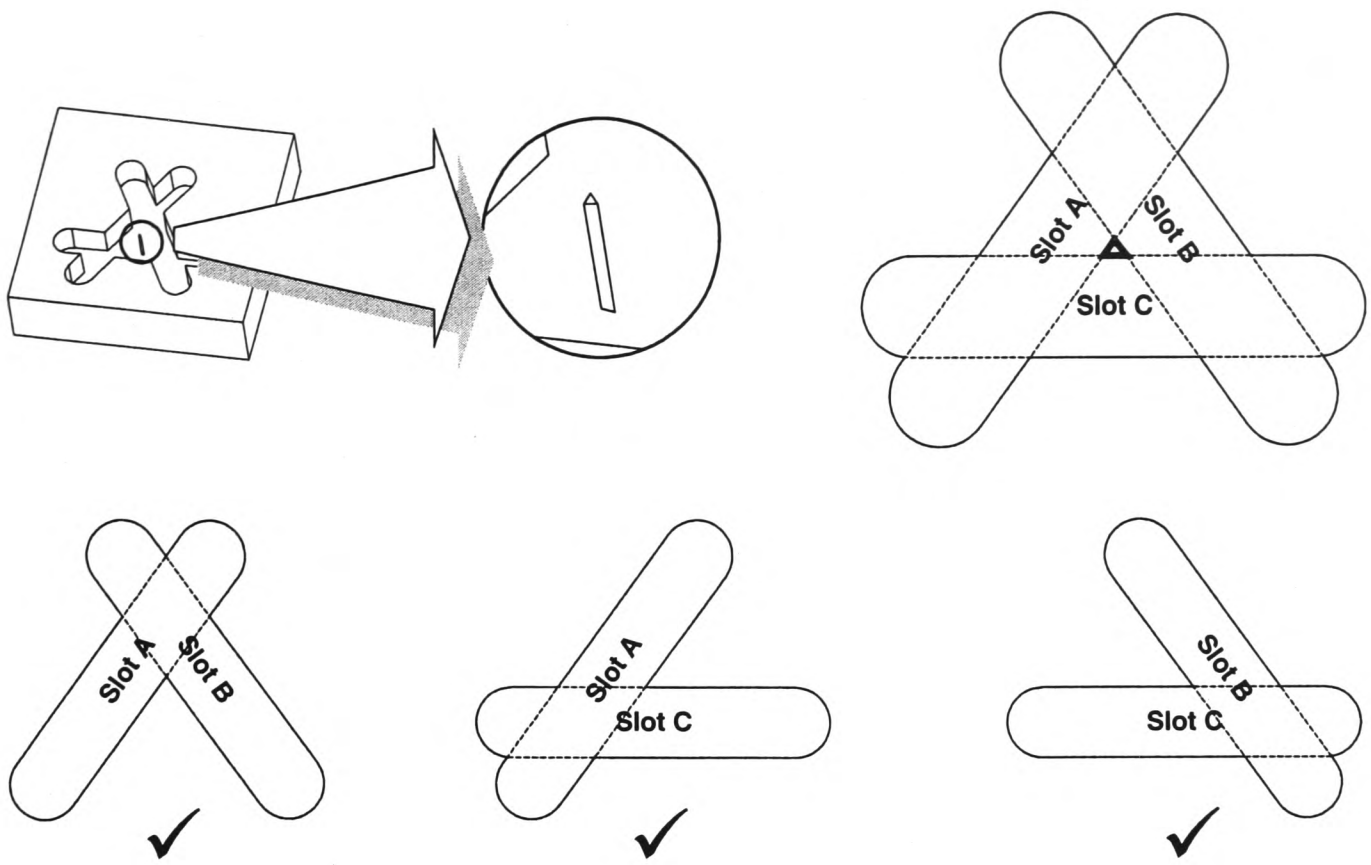


Figure 7-4: Thin section between three features

Some manufacturability problems involving three, or more features, remain undetected by MADSfm however. A specific case of thin wall is particularly obvious and shown in Figure 7-4, which reveals how three or more negative features can create thin walls, which eludes detection by the agent community.

Slot A, B and C are positioned in a triangle inside a block so as to create a thin central island. This thin section, which compromises part manufacturability, goes completely undetected by all feature agents. The detection method used by features is based on geometric beliefs concerning other individual features. As shown in Figure 7-4, intersecting slots taken by pair pose no manufacturing problems. From each slot's point of view, there exist two other negative features that intersect without creating problems. No mechanism is yet implemented inside MADSfm for detecting this type of partial contributions and the thin section is undetected.

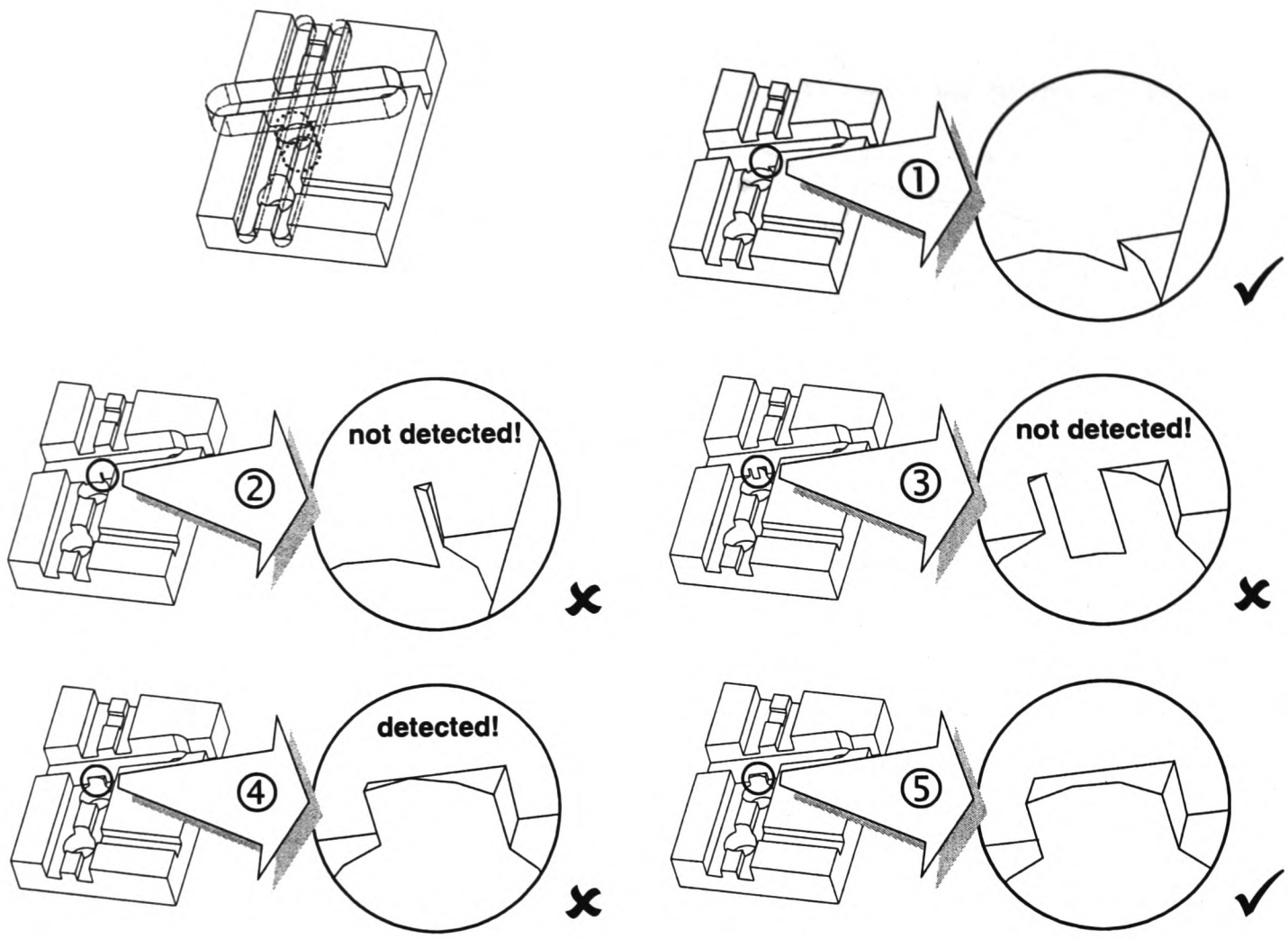


Figure 7-5: Undetected proximity problem involving three features or more

Figure 7-5 uses a well-known test component for feature recognition, borrowed from **[2]**, to show examples of undetected proximity problems inside MADSfm. The thin sections exhibited in Figure 7-5 are similar to that just detailed in Figure 7-4. The negative features of interest in the part are presented at the top-left. In order to create potential proximity problems inside the model, the designer modifies the position of the through hole at the centre of the part by translating it along the axis of the nested slot (Y-axis). The critical area of the design is shown magnified for each part configuration. The initial state, Figure 7-5 ①, presents no manufacturability problems. By moving the through hole by one millimetre in the Y direction, Figure 7-5 ②, a thin section is created that goes undetected by all agents concerned. In Figure 7-5 ③, the hole has been translated by another millimetre along Y and two thin sections are created that also remain undetected. An additional one-millimetre translation of the hole along Y, Figure 7-5 ④, return the model into a supported proximity configuration (two features only) that allows detection by the features involved. Figure 7-5 ⑤, show the component returned to a manufacturable configuration by a final one-millimetre translation of the hole along Y.

This limitation is the result of the simple agent interaction protocols used in the system. Indeed, MADSfm only supports interaction between two agents at any given time, which puts such problems out of reach. It is believed that the addition of more complex interaction models allowing co-operation between more than two features would eliminate the issues described above.

7.2.2 Automatic solving

Several interesting issues relating to automatic solving arose while experimenting with MADSfm in real design conditions. The most important two are addressed in the following sub-sections.

7.2.2.a Dynamic behaviour selection

Dynamic selection of a feature's behaviour (see section 6.3.4.g) is a novel aspect of design introduced by the adoption of autonomous agency inside the CAD system. It is an important aspect of using feature agents to assist the user. It defines what part of the design activity is delegated to agents and how it should be accomplished. This responsibility rest squarely on the designer's shoulder who must use behaviour selection to capture the design intent accurately so as to override any conflicts between his design experience and the limited knowledge of the feature agents. It was found that this activity is of great importance in order to reap the full benefits of the agent-based approach. Indeed, this activity determines the kind of corrective action that feature agents should undertake on behalf of the user. Experience has shown that wrongly selected features' behaviours can create an active model, which is not useful to the designer. Worst still, "badly behaved" models can work against the designer.

At present the designer is asked to control the individual behaviours of the agents. This requires a detailed understanding of the particular system. It may be possible to present the user with a set of functionally oriented controls (such as preserving through holes, etc.) that in turn trigger or disable particular behaviours. This in turn allows the user to concentrate on the functional aspects of the design without requiring detailed understanding of the agent-based mechanism by which this is achieved. This approach has not been adopted in the prototype system where it is felt more important to understand the consequences of the individual behaviours and so complete control is allowed.

7.2.2.b Geometric validity of Features

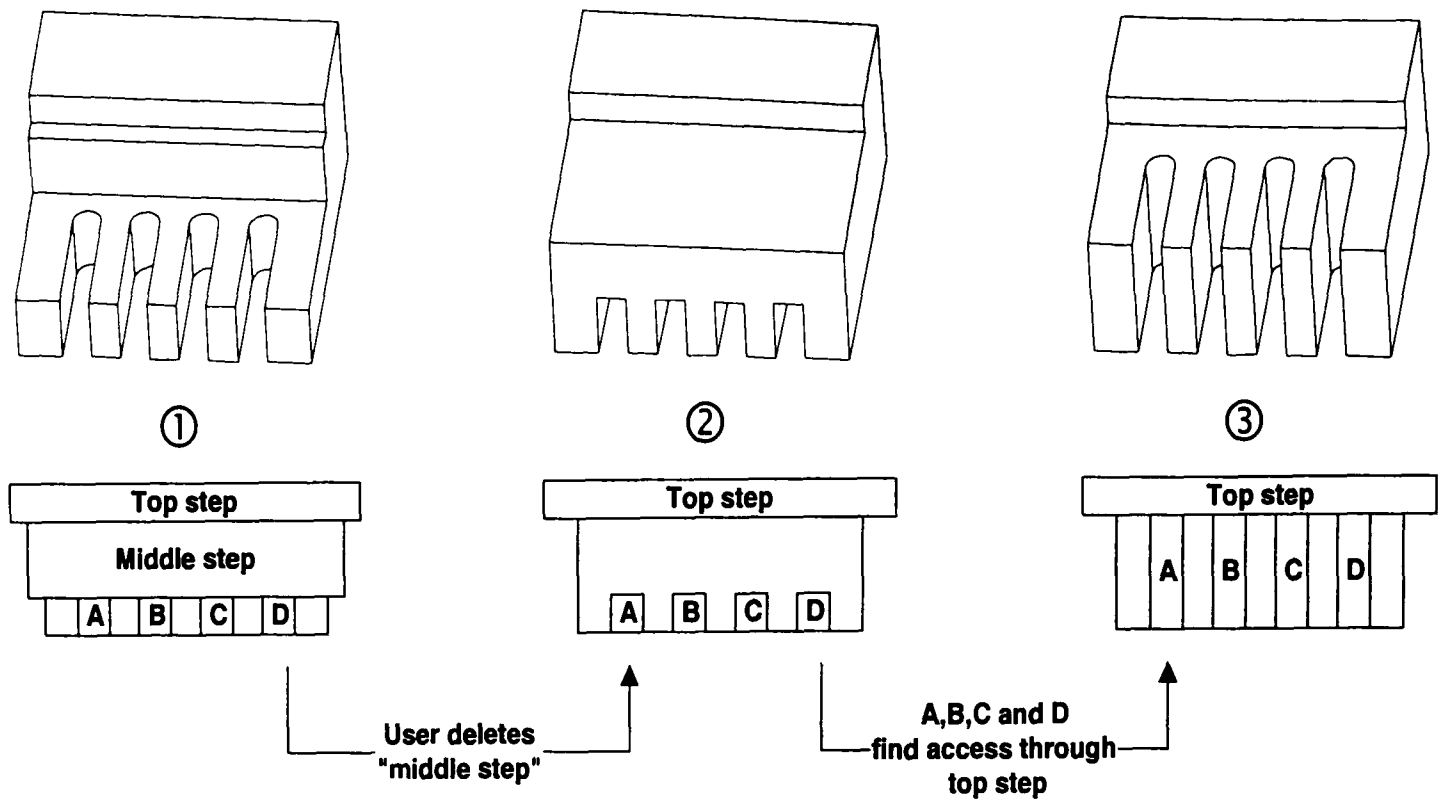


Figure 7-6: Self-correction creating invalid features

Given the feature agent's degree of autonomy, it is important to ensure their constant geometric validity. The capability of lightweight features remains limited and specific situations may arise that trigger undesired self-modifications. For example, the slots illustrated in Figure 7-6 are created as through slots. The designer might therefore select the access/minimality behaviour that preserves absolute depth of the feature (see section 6.3.4.c). A modification made to features above the slot results in a behavioural response aimed at ensuring a possible tool access route. Because the slots *accessibility* behaviour preserves the absolute depth of their bottom faces, their aspect ratio is distorted beyond the capability of the machining process they represent (see sections 2.2.2 and 3.3.2.a). Figure 7-6 ③ show that after self-correction the depth/width ratio of the four through slots is too high and conventional milling would not permit their machining.

The current system imposes no limitations on features' geometric aspect ratio after self-correction. This can result in features that cannot be produced using traditional machining methods. Although MADSfm currently provides no such mechanism, it appears clear that validation of the geometric aspect of features is needed to ensure they remain within the physical limitations of the production process.

7.2.3 Model stability vs. Agent autonomy

It is recognised that in engineering, the modification of existing models (re-design) is a more common task than design from scratch **[94]**. Engineers and designers tend to start from existing designs, adapting them to new situations, rather than starting from scratch. In this context, only portions of the design are actually modified while the rest should be preserved unchanged as much as possible. The preserved portions of the existing design are valuable because they represent validated partial solutions.

Feature autonomy is a major contribution of agent technology applied to feature-based design. It allows delegation of repetitive tasks to autonomous agents working on the designer's behalf. However, the desired autonomy might conflict with the need for model stability. A compromise between autonomy and stability must be struck. Features should be allowed a degree of geometric freedom that allows them to solve common problems on behalf on the designer. The self-induced modifications should, nonetheless, remain localised to problematic areas and not spread uncontrollably through an entire design.

7.2.3.a Livelocks

Livelocks (see section 6.2.1.a) are situations in which agent activity is stuck in a never-ending event loop. In the case of features inside a mechanical design, livelocks have potential harmful effects because they an endless cycle of geometric changes that might compromise global model stability.

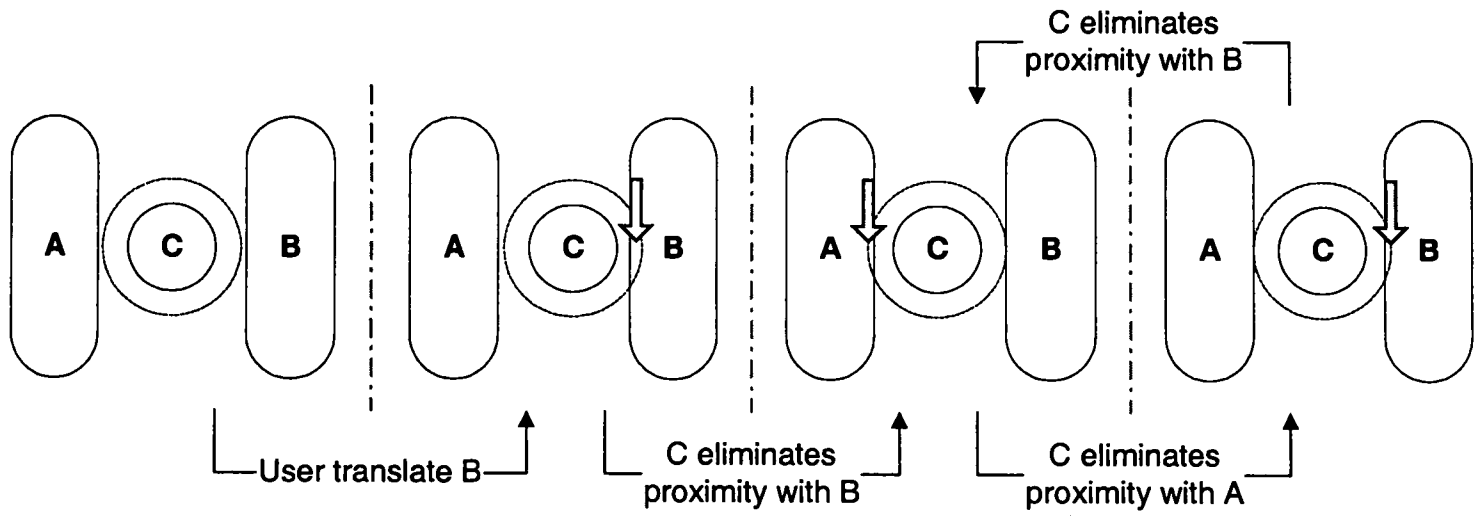


Figure 7-7: Livelock situation example

Figure 7-7 shows a livelock situation arising between two slots and a hole. Slots A and B have their proximity behaviour disabled, but C has not. By translating B without enabling its

proximity behaviour first the user create a situation in which C livelocks while attempting to eliminate thin sections.

A simple livelock detection mechanism is implemented within MADSfm that monitors the time during which agents actively modify the design (see section 6.3.3.c). Long periods of activity that do not lead to stable solutions hints at potential livelocks. In such cases the MAS is able to show self-restraint by stopping its central activity schedule, which can preserve partial solutions until user intervention.

This minimal implementation of livelock detection has proven its ability to detect livelocks when they occur inside models. The automatic shut down of agent activity prevents accidental loss of partial solutions and allows user intervention on the model. The latter can manually force a solution or restore the model to a previous stable state before unfreezing agents. However, it can not (yet) identify which particular agents caused the livelock and must stop all activity to preserve the design from potential harmful self-corrections. Moreover, some long period of useful, productive agent activity might be mistakenly detected as livelocks.

7.2.3.b Chain reactions / Snowball effect

In addition to the case of livelock, intense activity of the agent-based model can also arise from cascade reactions between interacting features. That is to say a chain of events can be generated between interacting agents that displace/propagate manufacturability problems within the model.

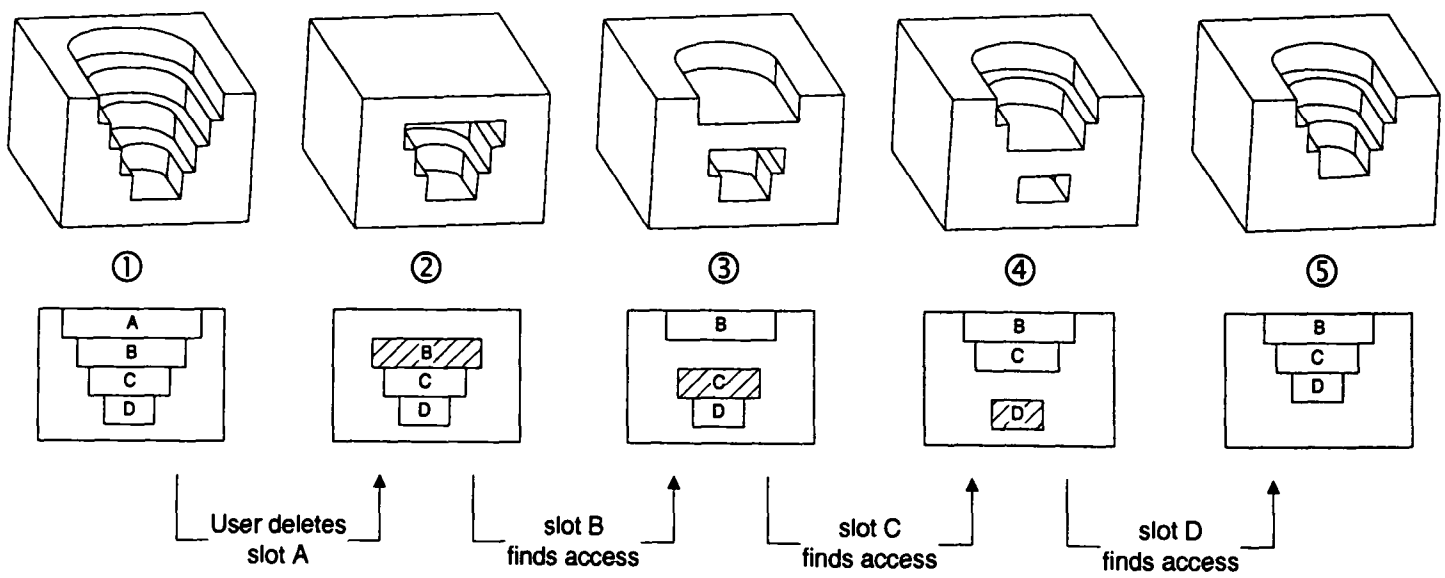


Figure 7-8: Chain reaction in solving nested feature's access

For instance, the simple case of nested features, illustrated in Figure 7-8, present a typical configuration than creates chain reactions while solving access problems. An initial model made of four nested features within a block, in Figure 7-8①, contains no manufacturability issues. By deleting the feature A (top feature) the user renders feature B inaccessible. Consequently this makes features C and D inaccessible also, since a link of their indirect access route disappeared. However, because of the locality of agents (see section 7.2.1.a) only feature B detects an access problem. The application of its solving knowledge results in the situation shown in Figure 7-8③, where feature C is now the only one detecting an access problem. The access problem has therefore been transferred from one feature to the next, creating a chain reaction. Within the described chain reaction, further behavioural responses of the system leads to Figure 7-8④ then Figure 7-8⑤.

Chain reactions are consequences of the locality of feature activity. An example of a positive chain reaction was shown in Figure 7-8. However, depending on the type of behaviour involved, such chain of events can have undesired consequences on the model's geometry. In particular, it should be noted that a major difference exists between Z-oriented and XY-oriented behavioural responses, which is related to predictability of consequences and reversibility of changes.

The uni-dimensionality of Z-oriented chain reactions means that they are generally reversible. Indeed, within the 2½D space used for MADSfm, modification along Z consist of ordering and resizing Z occupancy intervals (see section 6.3.2.b). Whether a single change or a multitude of changes occurs along Z, the original situation can always be re-obtained through a reordering/resizing of Z-intervals. In contrast, multiple transformations occurring within XY planes are not easily reversible unless a full history of previous states is maintained within each agent.

Chain reactions are tightly linked to the principle of locality discussed in 7.2.1.a. They are a manifestation of a global system's behaviour emerging from local interactions taking place autonomously at agent level. In this respect, they are a desired quality for a MADS that ultimately leads to its usefulness. However, in the engineering context of designing mechanical components, caution must be exercised when reaping the benefits. Model stability and autonomy must adequately balance each other so as to provide assistance to the user without becoming a nuisance or liability.

7.2.3.c Geometric modification and initial solving

Two courses of action are possible after a feature is modified by a user request that trade off model stability versus designer's intent.

- The first option is for the modified feature to analyse then apply its solving knowledge before broadcasting its geometry. This approach gives the modified feature a chance to minimise its impact on the existing model. By applying its solving knowledge immediately after modification, the system emphasises model stability over the latest modification.

- In the second option, modified features broadcast their geometry without preliminary solving, which may trigger various behavioural responses from neighbouring features. The user's intent is therefore given priority over the model's current state.

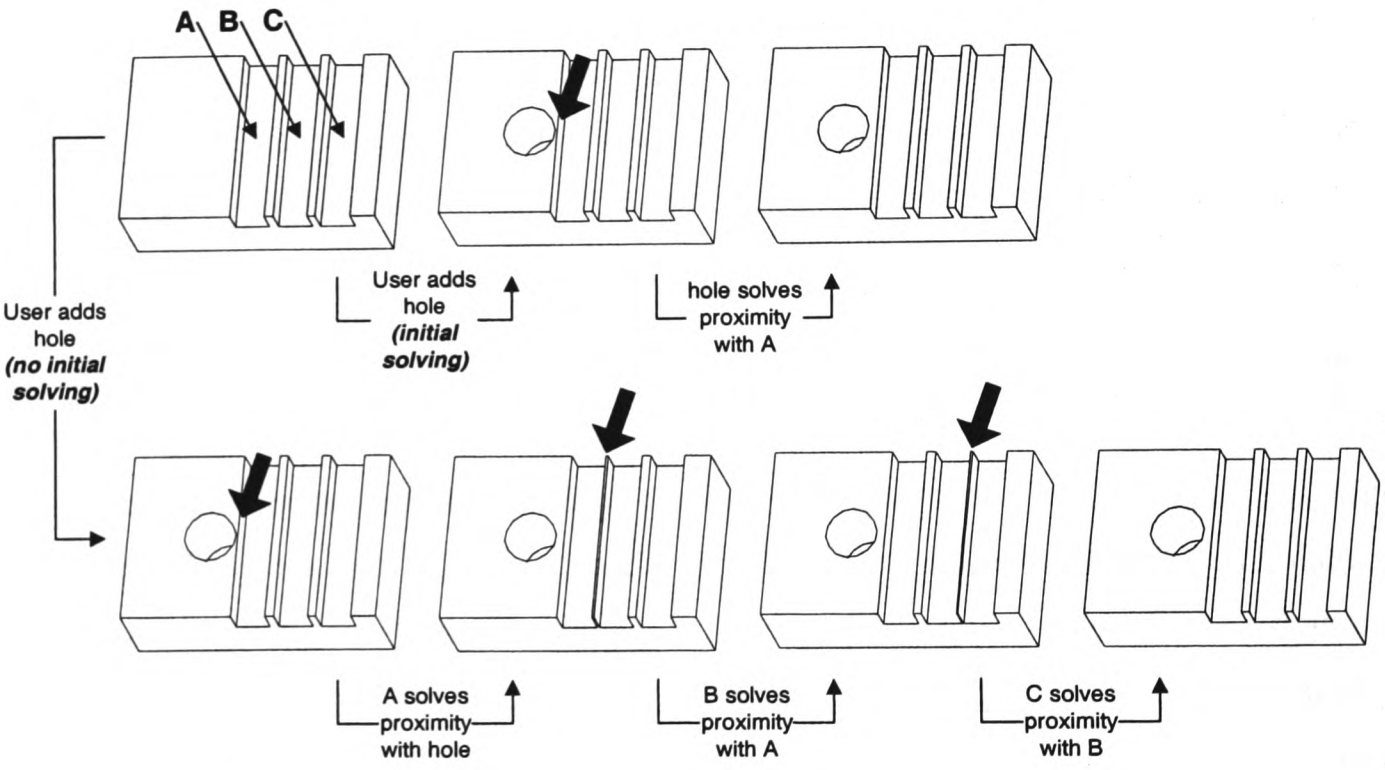


Figure 7-9: Influence of initial solving after changes

Figure 7-9 illustrates how both options yield different results when adding a hole near an array of slots. In the top sequence, initial solving allows the pocket to solve a proximity problem, hence preserving the rest of the model in its stable state. In the bottom sequence however, immediate broadcast of the pocket's geometry result in a chain reaction that propagates through the array of slots before resulting in a new stable state.

Experience has shown that the conservative first option is often preferable from the designer's point of view because it limits unforeseen consequences of requested geometric

changes. Moreover, the user can always confirm his design intent by manually undoing any self-correction performed by the new feature.

7.3 Other functions

7.3.1 Part files

The current interface of MADSfm allows interactive construction of feature-based parts, which generate the living agent community representing these parts. Computing resources are never infinite and one cannot expect models to remain active at all times. However, 4.6.4 discussed the difficulties that agent serialisation poses. It is also evident that the ability to use feature-based models not created using agents would be of great benefit to users. A mechanism is therefore needed, that permits the static storage of agent models and their restoration as a living community of autonomous agents. This mechanism should also cater for passive feature models using compatible a feature set.

MADSfm permits the saving of agent-based model inside static text files containing only the essential geometric and behavioural information required to rebuild the part. As discussed in 4.6.4, saving agent-based parts in such files entails the loss of important information. Indeed, none of the derived geometric data and dynamically generated beliefs are preserved during this serialisation scheme. This problem can be overcome though, since agents can autonomously generate this information concerning their local environment at runtime. However, this learning process (see section 6.3.4.f) must be allowed to happen before the agents inhabiting the model can perform their tasks properly.

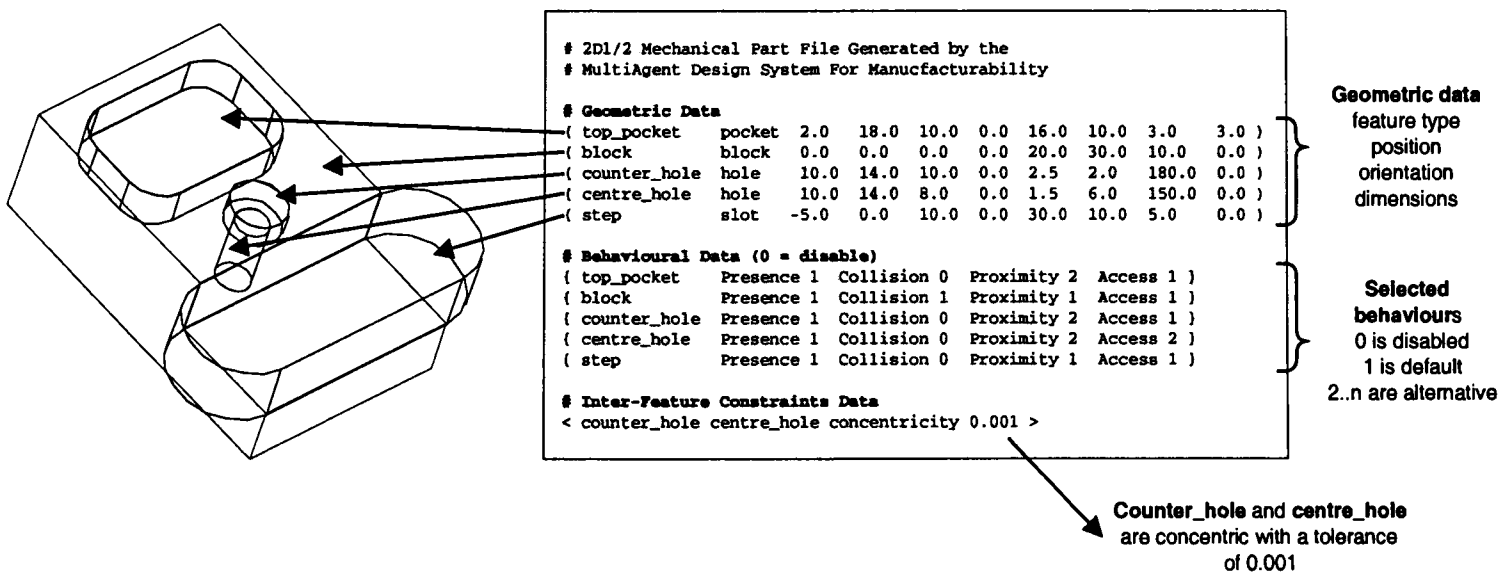


Figure 7-10: Example part and its description file

Figure 7-10 illustrates a sample part-file and the 2½D component it represents. The file is made of three separate sections containing

- the geometry of individual features,
- the behavioural preferences of individual features,
- and the description of any existing inter-feature constraint.

7.3.2 Inter-feature constraints application

The ability to express inter-feature geometric constraints is an important aspect of CAD systems used in the production of complex models. Inter-feature constraints allow the designer to create *rules* that express high level geometric properties concerning a modelled part. Relative positioning and orienting of feature, in particular, can greatly increase the ease of use of a feature-based design system.

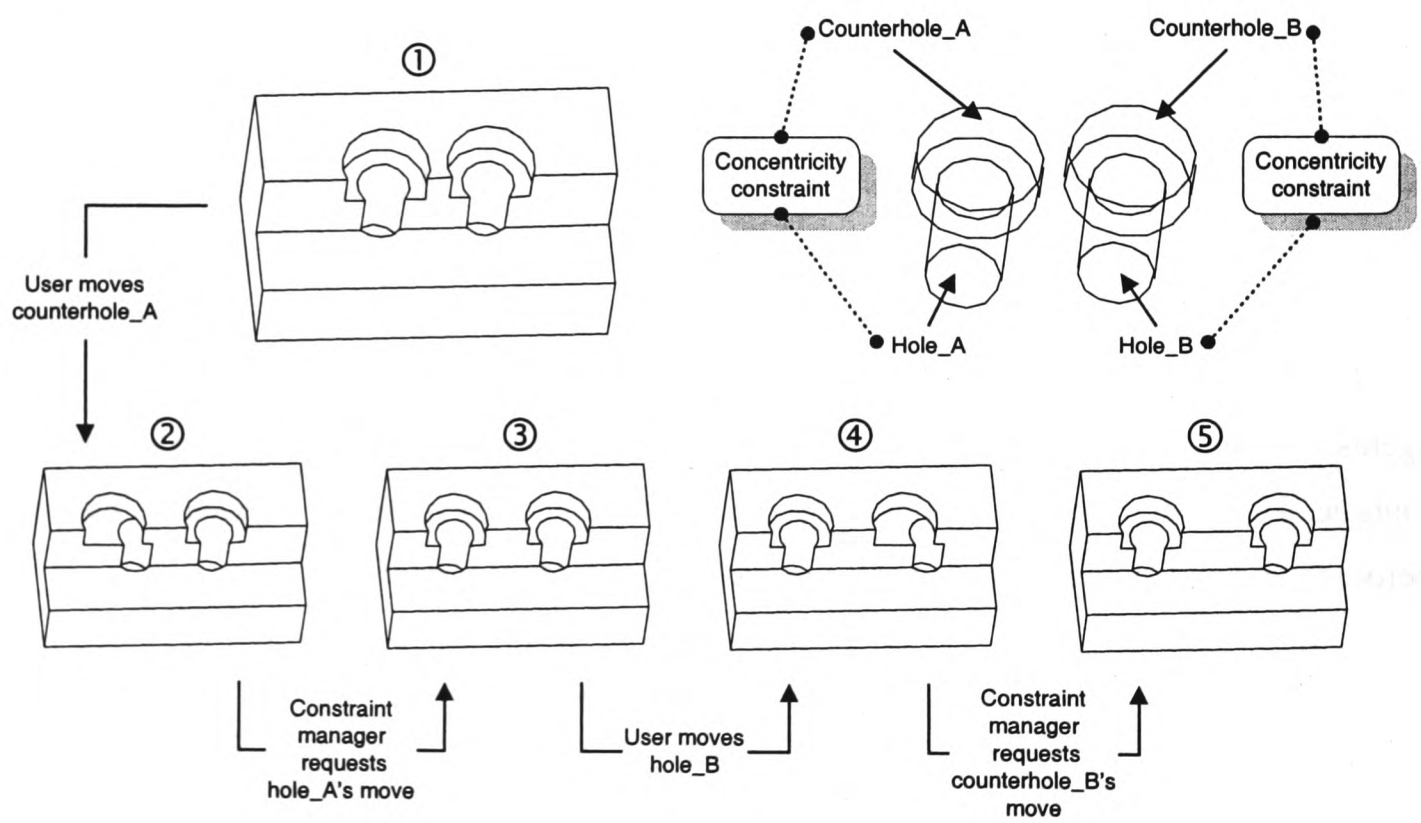


Figure 7-11: Concentricity constraints between hole features

MADSfm includes a primitive constraint solver that demonstrates the feasibility of mixing user defined constraints with feature agents' desires. A constraint manager agent (described in 6.3.3.c) is included in the system that can handle basic bi-directional geometric constraints between features. Figure 7-11 demonstrates how two concentric holes can be constrained as such. The constraint manager agent monitor geometric changes in the system. When a feature participating in a constraint is modified, the manager checks the constraint

satisfaction rule. If required, it applies the constraint's solving procedure, which typically result in a request to the unmodified feature participating to the constraint.

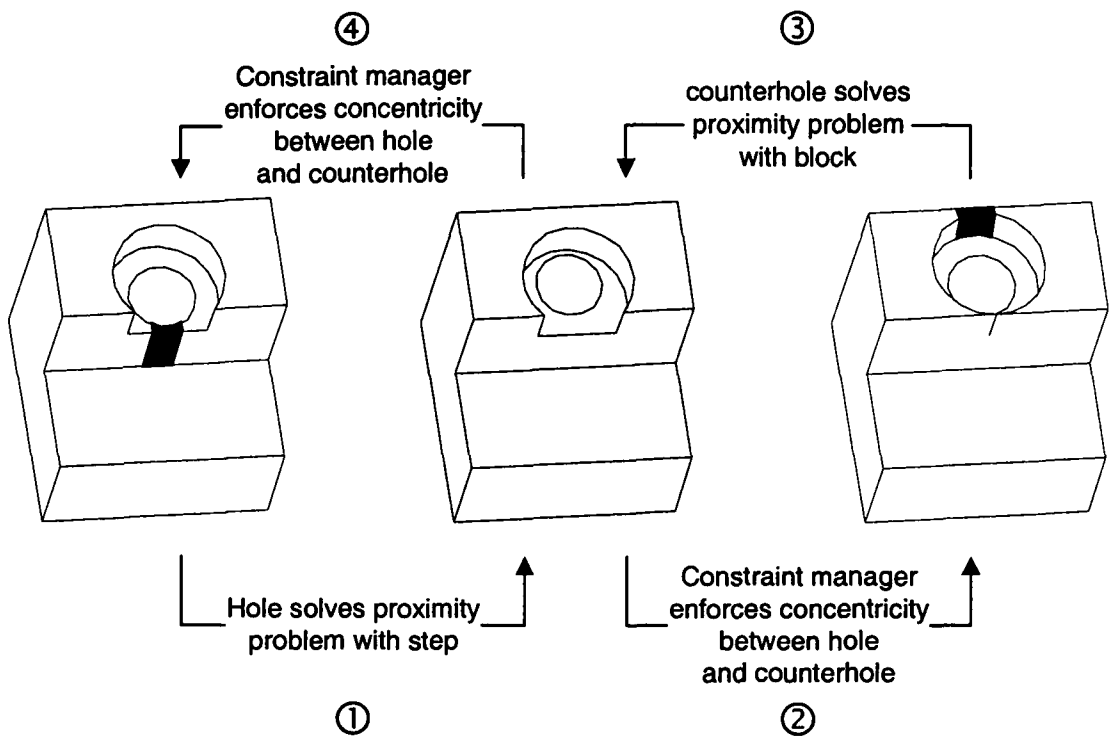


Figure 7-12: Livelock caused by concentricity constraint

This basic mechanism demonstrates that bi-directional constraints between agents can be enforced. However, it does no extra checking before requesting geometric modifications, which can easily lead to livelock situations. Indeed, as show in Figure 7-12, the combination of proximity behaviours and a concentricity constraint may livelock the system.

7.3.3 Process planning hints

The activity of feature agents is shared between real-time geometric analysis and autonomous correction of the design. These inter-connected tasks are performed over-time (see section 5.3.4) by features that need to learn about their local environment in order to operate efficiently. Indeed, the detection of potential problems and their autonomous correction relies on the local knowledge generated (and stored) by feature agents during their life inside the model. This accumulated data concerning neighbouring features represents a great deal of useful knowledge for process planning. In particular, the accumulated beliefs relating to tool access can be used to provide valuable hints to a process-planner concerning the sequencing of operations required to manufacture a component.

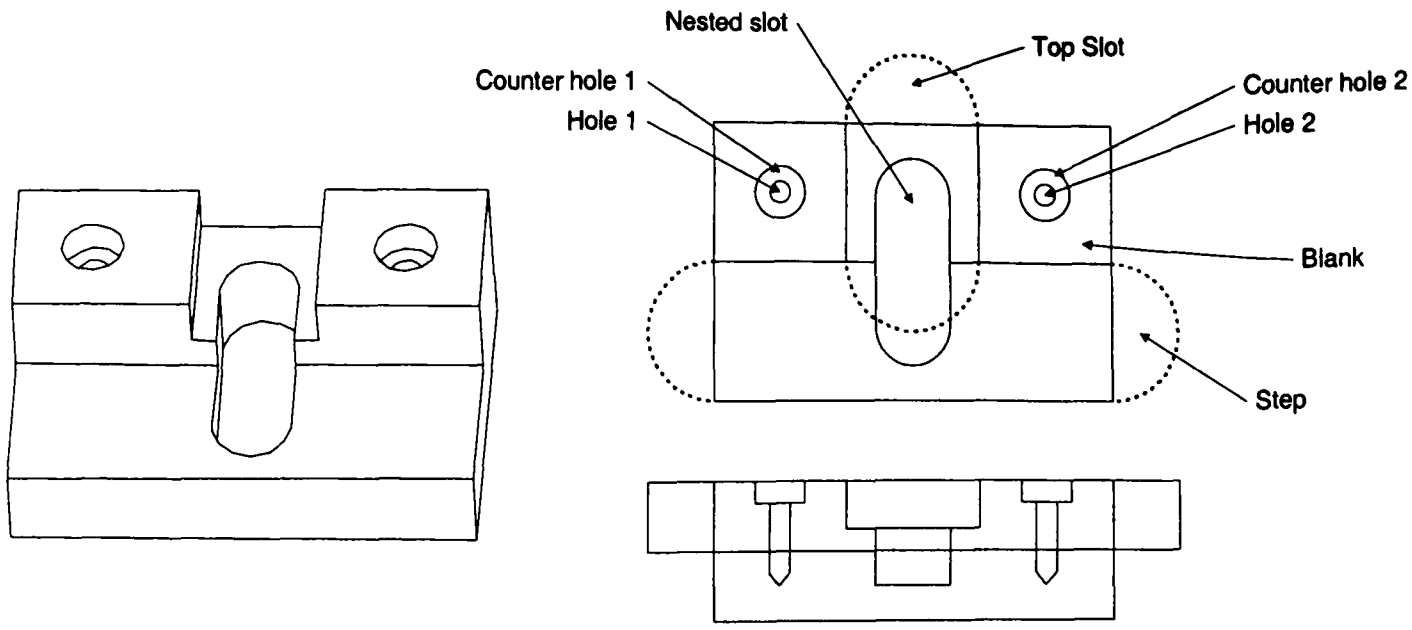


Figure 7-13: Example part

```
# Geometric Data
( blank         block 0.0   0.0   0.0   0.0   30.0  20.0  10.0   0.0 )
( step          slot -6.0   0.0   10.0  0.0   42.0  10.0  5.0    0.0 )
( counter_hole1 hole  5.0   15.0  10.0  0.0   2.0   2.0   180.0  0.0 )
( counter_hole2 hole  25.0  15.0  10.0  0.0   2.0   2.0   180.0  0.0 )
( hole2         hole  25.0  15.0  8.0   0.0   1.0   5.0   150.0  0.0 )
( hole1         hole  5.0   15.0  8.0   0.0   1.0   5.0   150.0  0.0 )
( slot1         slot  19.5  5.0   10.0  90.0  20.0  8.0   3.0    0.0 )
( slot2         slot  18.0  2.0   7.0   90.0  15.0  5.0   5.0    0.0 )

# Behavioural Data
{ blank         Presence 0 Collision 1 Proximity 1 Access 1 }
{ step          Presence 1 Collision 0 Proximity 1 Access 1 }
{ counter_hole1 Presence 1 Collision 0 Proximity 2 Access 1 }
{ counter_hole2 Presence 1 Collision 0 Proximity 2 Access 1 }
{ hole2         Presence 1 Collision 0 Proximity 1 Access 2 }
{ hole1         Presence 1 Collision 0 Proximity 1 Access 2 }
{ slot1         Presence 1 Collision 0 Proximity 1 Access 1 }
{ slot2         Presence 1 Collision 0 Proximity 1 Access 1 }

# Inter-Feature Constraints Data
< hole1 counter_hole1 concentricity 0.0 0.001 >
< hole2 counter_hole2 concentricity 0.0 0.001 >
```

Listing 7-7: Bare input file for example part

```
# Process Planning Data
[ counter_hole2 --before--> hole2 ]
[ counter_hole1 --before--> hole1 ]
[ step --before(*)--> slot2 ]
[ slot1 --before(*)--> slot2 ]
[ hole1 --same_setup--> counter_hole1  ]
[ hole2 --same_setup--> counter_hole2  ]
```

Listing 7-8: Process planning hints generated by agents

To demonstrate how much useful information feature agents produce during their normal activity, one can compare the data input provided by a designer and the file output produced

by MADSfm. During a typical design session, the user feeds geometrical information and behavioural preferences to the system. He can also create geometric relationship between features. MADSfm can capture this user input inside an input file. This file contains all the data provided by the user concerning feature geometry, behavioural preferences and the description of geometric constraints as shows in Listing 7-7. Loading this file into MADSfm creates the living model illustrated in Figure 7-13. Through their autonomous activity, features inside this model accumulate information useful to process planning. Indeed, a few seconds after loading this file, the agent community has enriched it with precedence constraints as demonstrated in Listing 7-8. These constraints result from the validation of tool accessibility by individual feature agents and from the analysis of expressed geometrical constraints between features.

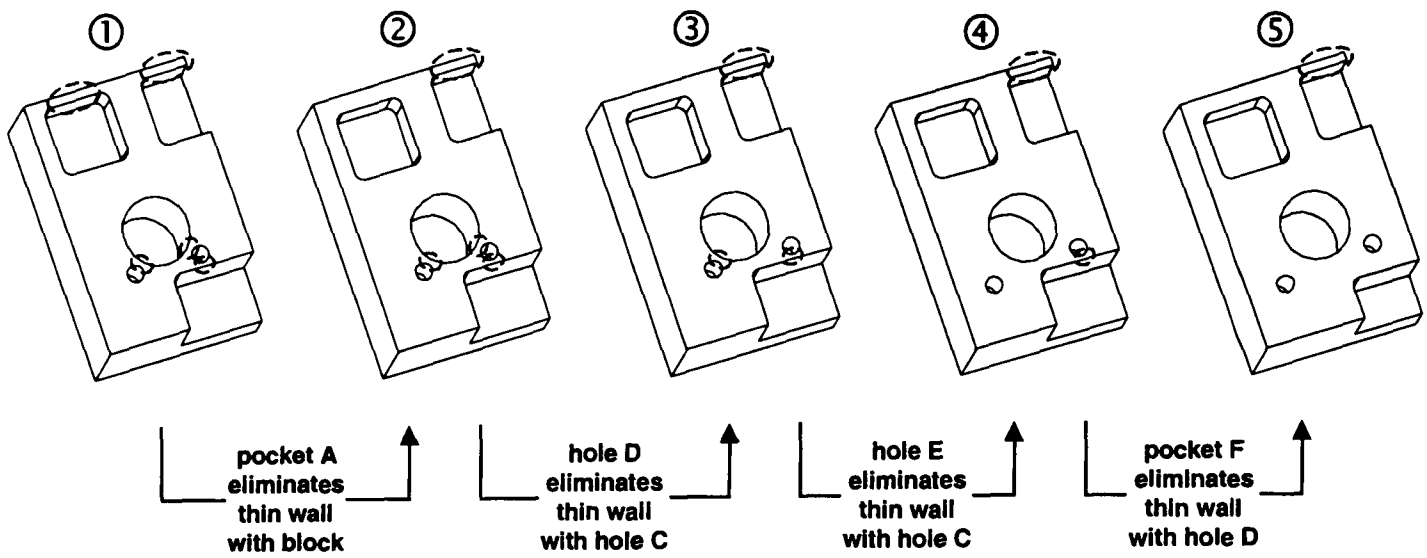
- A full indirect access of A through B translates into a strict `[B-->A]` constraints that signify B must be machined before A.
- Indirect access obtained through the combination of partial contributions (see section 6.3.4.b) translate into a list of `[X-->Y]` constraints. The `(*)` signifies that a number of features Y must be machined before X, but the order in which they are machined is not important.
- Inter-feature geometric constraints between features can translate into `[X--same_setup-->Y]` rules depending on the expressed tolerance. Where `same_setup` expresses the fact that the desired tolerances can only be obtained if both operation are performed on the same machining setup.

A process planner can use these constraints generated by MADSfm to determine possible machining sequences for the part. More importantly, because they result from the autonomous activity of features, they are readily available at any stage during the design process.

7.4 Test Components

A number of test components are presented that demonstrate behavioural responses to individual validation criteria that were previously described in 6.3.2.d. Other components are presented that show important properties of MADSfm. All tests were performed on a PC compatible equipped with an Intel® Pentium® II 350MHz processor, 128Mb of memory, and running Microsoft® Windows NT® 4.0 workstation.

7.4.1 Proximity test



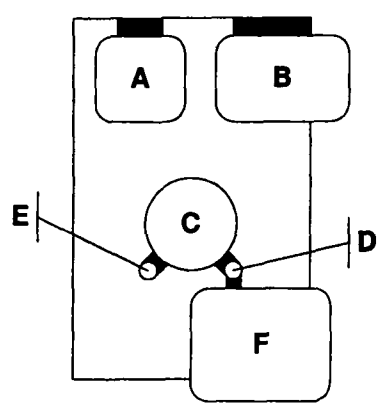


Figure 7-14: Example of proximity solving

Figure 7-14 demonstrates the proximity solving capabilities of MADSfm. A test component that contains different types of thin sections is created inside the system while agent activity is disabled. Shown in Figure 7-14①, it contains five different cases of thin section. Pocket A creates a thin wall with the block. Pocket B also has a proximity problem with the block although their profiles are partially intersecting. Holes D and E both create thin sections with hole C. Finally proximity also exist between pocket F and hole D.

The agent activity is started and the model performs self-corrections. The entire test takes under 2 seconds to complete and requires the exchange of around 70 KQML messages. The proximity between pocket A and the block is handled gracefully with A translating itself away from the block's edge. However, the very similar thin wall created by pocket B and the block remains unsolved. This is due to the fact that B's profile intersects with the block's profile, which renders the determination of a minimal inter-feature distance useless.

Hole E eliminates a thin wall with C with a simple avoidance vector. Hole D on the other hand demonstrate the use of a damped escape vector (see section 6.3.4.c) that ensures D does not "enter" pocket F while attempting to solve a proximity problem. Indeed, the type of

intersection between D and F is maintained, which important to preserve the design intent of the user.

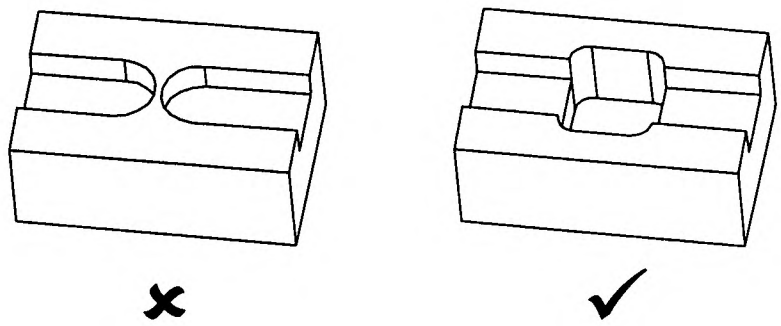


Figure 7-15: Example of false proximity

MADSfm proves able to handle thin sections between negative features as well as between negative and positive features. Limitation remains is the current prototype however. The inability of solving proximity between partially intersecting features was shows in Figure 7-14. It should also be noted that proximity detection does not currently check that the thin section takes place within a positive feature and can therefore trigger false proximity alarms as illustrated in Figure 7-15 where the addition of a pocket eliminates a potential thin section between slots.

7.4.2 Access test

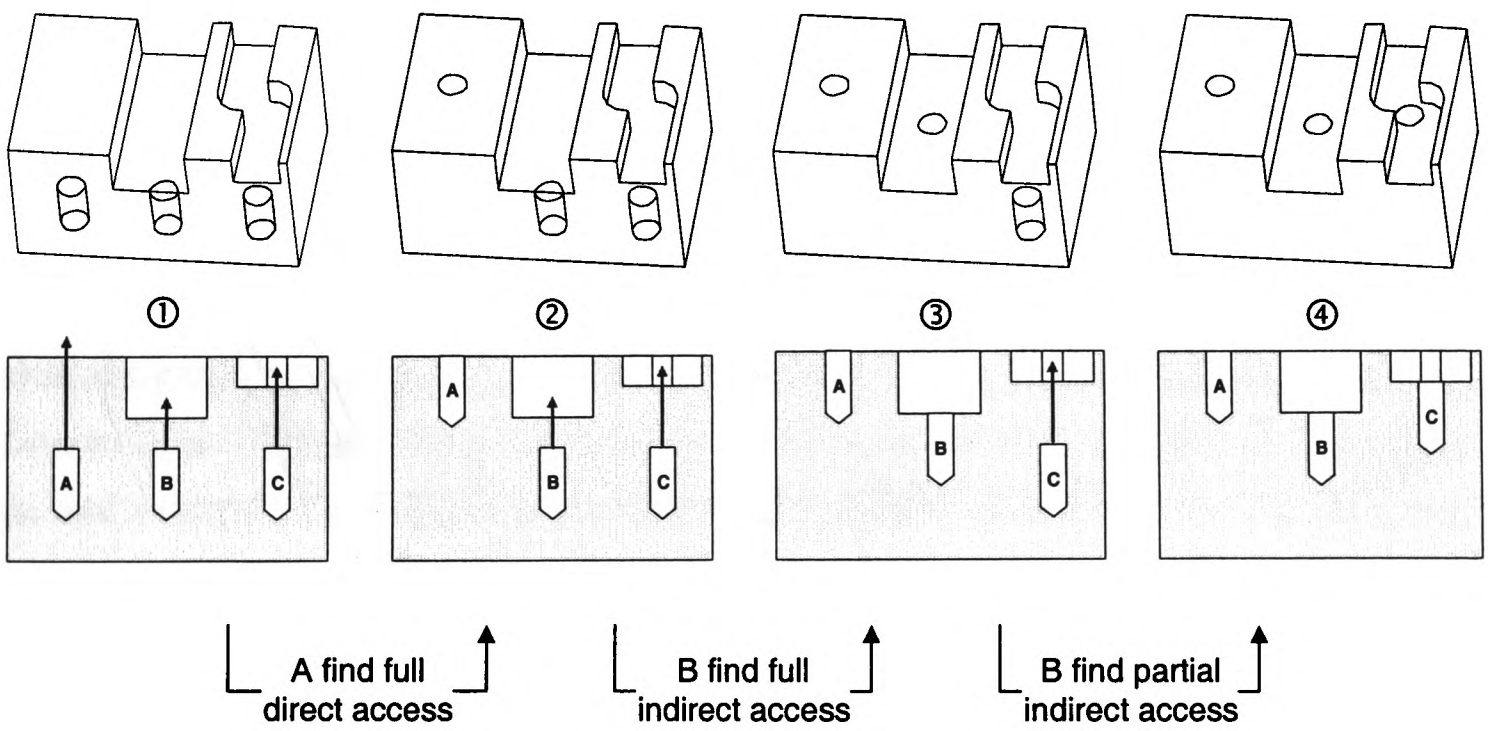


Figure 7-16: Example of access solving

Figure 7-16 illustrates the typical outcome of negative features applying the access solving behaviour provided in MADSfm. Three hole A, B and C are added inside an existing

component in a way that does not allow tool access for their machining. Each hole demonstrates a different type of access route found by the access behaviour. Hole A finds direct tool access by placing its top face at the top of the blank. Hole B, obtains access through another feature whose XY profile surrounds it. Finally, hole C obtains indirect access through the combination of two other features. The entire test takes under 1 second to complete and requires the exchange of around 50 KQML messages.

7.4.3 Collision test

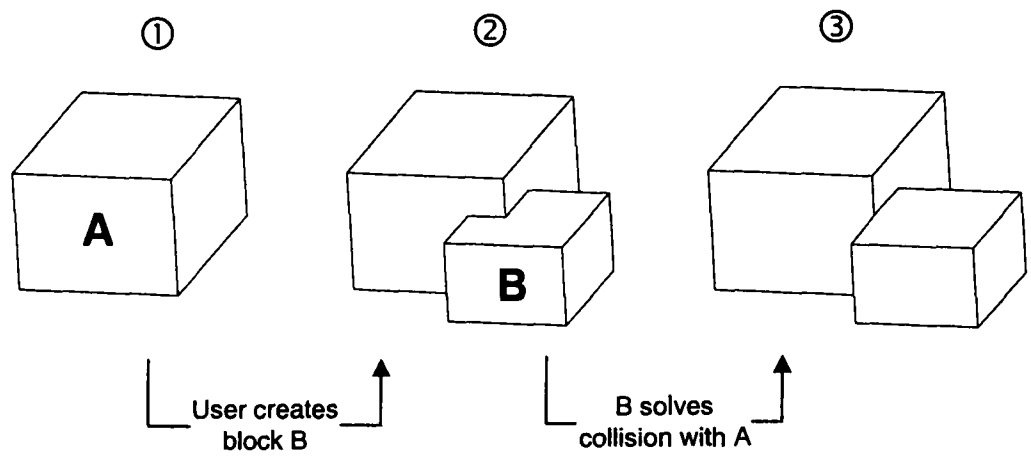


Figure 7-17: Example of collision solving

The only collision behaviour currently implemented in MADSfm concerns positive features. Indeed, as shown in Figure 7-17, block B ensures it does not share the same physical space as block A by taking evasive action. After the creation of block B, this test completes almost instantaneously.

7.4.4 Presence test

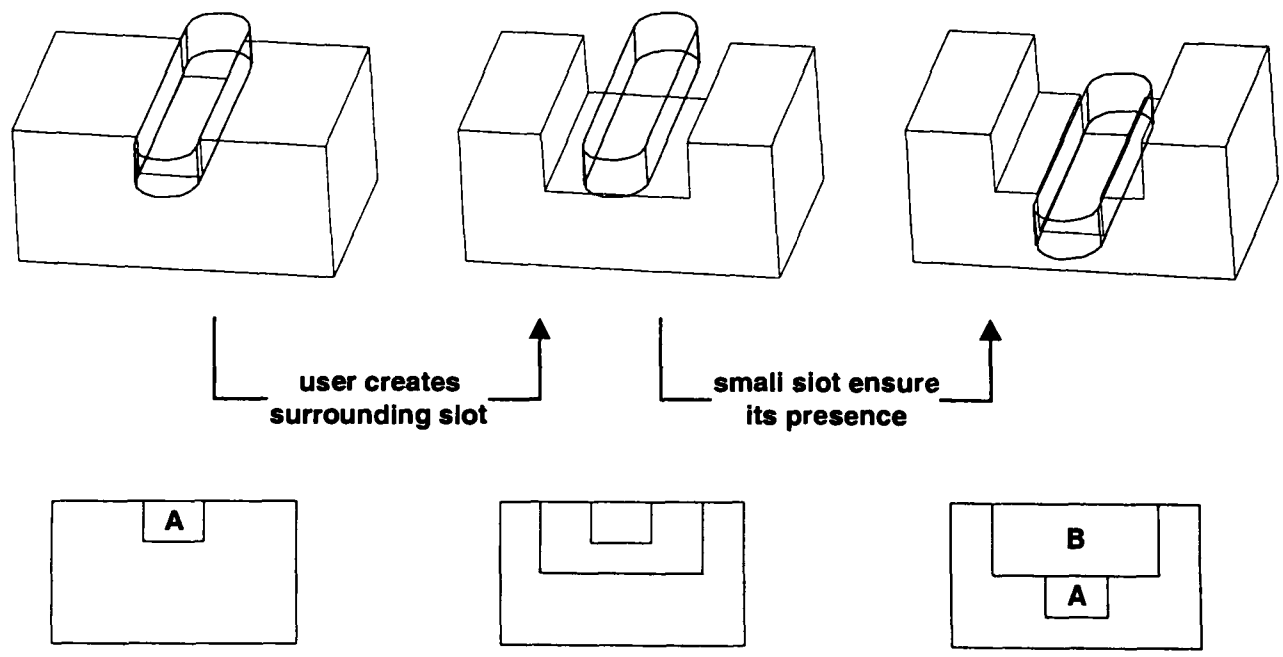


Figure 7-18: Sequence showing presence solving

Figure 7-18 shows the negative features A ensuring its contribution to the finished component. Negative features that detect they are fully intersecting with another negative feature (or the union of several features) attempt to find a new Z position that allow their presence inside the designed part. After the creation of slot B, this test completes almost instantaneously.

7.4.5 Minimality test

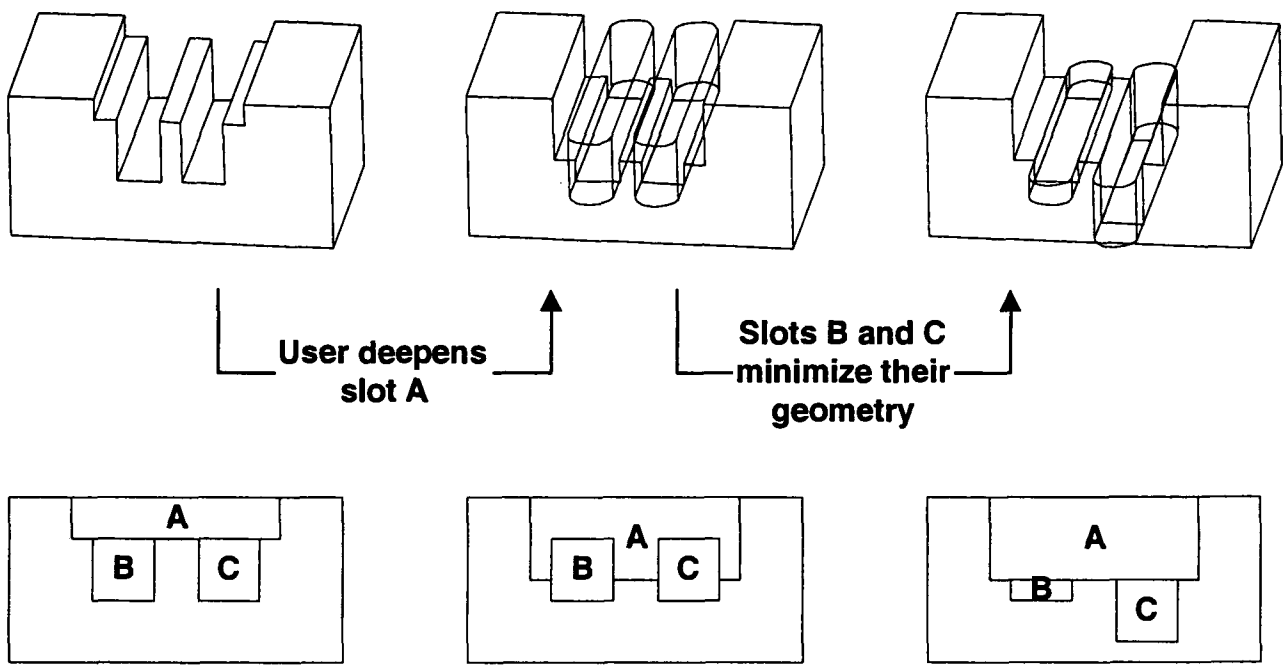


Figure 7-19: Sequence showing minimality solving

Figure 7-19 illustrates the minimality behaviour of two slots. By changing the depth of top slot A, the user renders slot A and B non-minimal. That is to say that part of the feature geometry will not translate into actual cutting of material from the blank. It should be noted that the minimality behaviour of features is consistent with their access behaviour. Indeed, feature preserving absolute depth at their bottom use scaling (see slot B in Figure 7-19) instead of translation as a course of action. For this reason, depending on the behaviour selection made by the user, slot B and C behaves differently in response to an identical minimality problem. Slot B preserves the absolute Z of its bottom face while slot C preserves its relative depth. The minimality test takes under 1 second to complete after the change on slot A.

7.4.6 Space partitioning test

The addition of a space partitioner service agent to the system allows great reduction in the communication load of the system when handling complex components. To measure the gain obtained, a component containing 18 features (illustrated in Figure 7-20①) is loaded in

MADSfm. The messaging activity is monitored while starting the agent activity until a stable configuration is reached. The following results are obtained:

- **Without** space partition: under 450 messages exchanged.
- **With** space partition: under 175 messages exchanged.

The significant reduction in what remains a moderately complex model demonstrate the importance in addressing the global communication load by providing a clustering mechanism as described in section 4.6.2. It should be noted that the gain obtained using space partitioning depends on the intrinsic geometric clustering of each model but is expected to increase with the number of features present.

7.4.7 Example of complex components

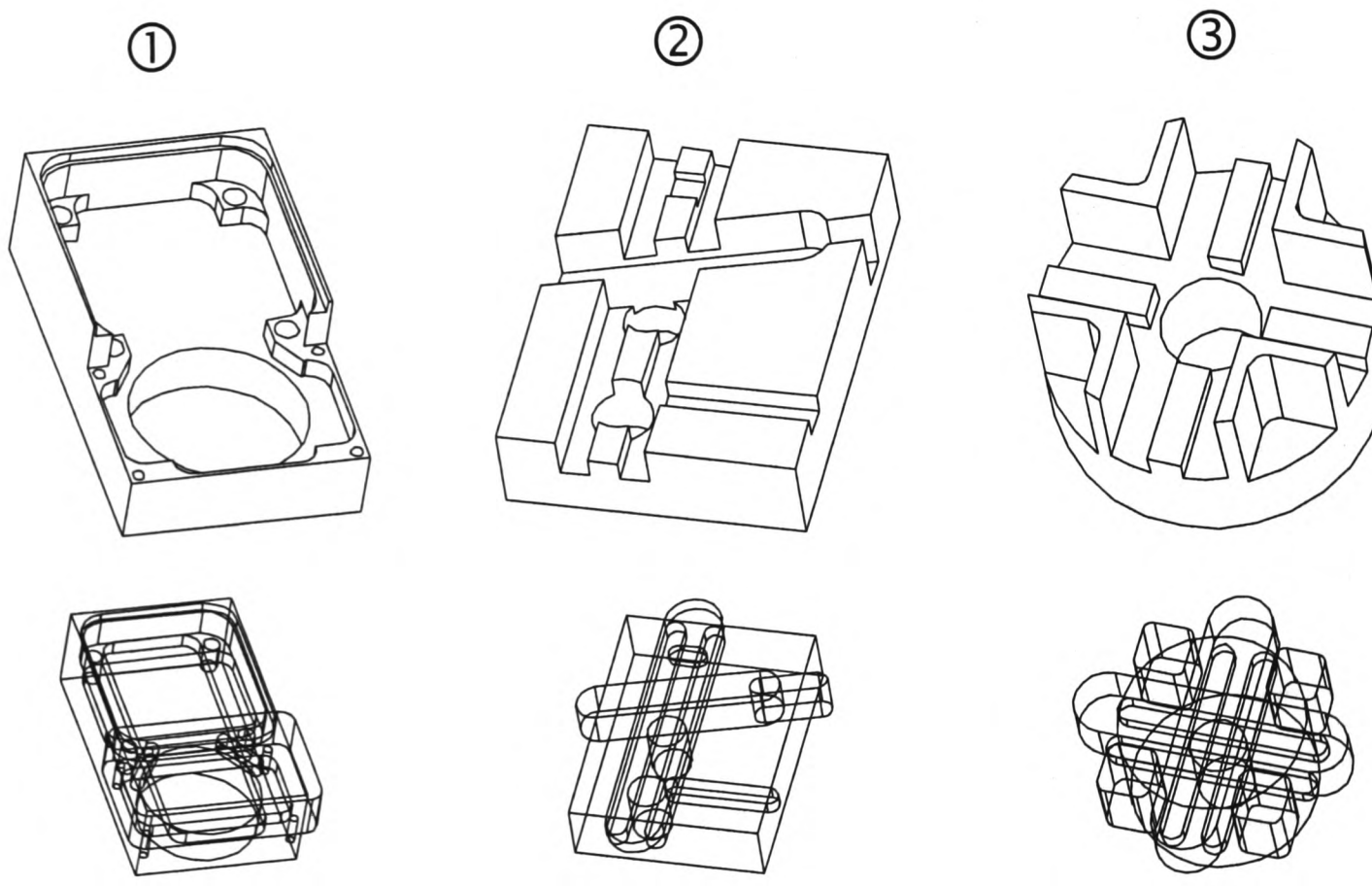


Figure 7-20: Example of modelled parts

Moderately complex components can be modelled using MADSfm (see Figure 7-20). The system scales gracefully with the number of features. This was expected because geometric locality is exploited in order to reduce the overall workload of features. However, the basic livelock-detection mechanism described in 6.3.4.d does not scale well. Indeed, when handling large models the activity-monitor agent struggle to differentiate between useful agent activity and potential livelocks. Larger models, containing numerous features may

generate long chain-reaction (see section 7.2.3.b) during modelling that can be misinterpreted for livelock situations.

7.5 Conclusion

MADSfm, the experimental implementation of a multiagent design by feature system was developed to a usable stage. Various tests and experiments were carried out to validate the validity of the concept of autonomous feature agents and to show its benefits and limitations. The individual behaviours of feature agents have been demonstrated and their shortcomings exposed.

Local activity of feature agents has proved its ability to generate a global emergent behaviour that brings benefit to the designer. The autonomous activity of features ensures their local manufacturability through real-time geometric analysis and application of solving knowledge concerning well-defined criteria. Although far from a real modelling tool, the implemented system demonstrates that a self-correcting feature model can be a useful feature for designers. Created by applying agent technology to feature-based design, the approach provides good performance and offers genuine assistance to the user in the area of manufacturability analysis. MADSfm shows excellent ability at solving access and minimality problems without undesired consequences on the rest of the design. Experience has shown that proximity and collision behaviours are more prone to slip-ups because they involve XY transformations that can't be easily reversed. The flexible behaviour selection mechanism permits the user to capture design intent in a novel manner. It also allows partial solutions to be safeguarded and agents decisions to be overridden.

Some limitations have also been described that are mostly due to the limited power of geometric representation and the simplicity of agent interaction protocols. In particular the inability to handle some problems involving more than two features demonstrate that more evolved agent interaction is needed for features.

Chapter 8

Conclusions

8.1 Agent technology for design and manufacturing

The manufacturing industry motivated by increasing competition strives to achieve shorter lead-time, better quality and lower costs. These major challenges can be partially won or lost during the very early stages of product design. The traditional gap between design and manufacture remains a major obstacle in this race against time because it often requires numerous design/analysis/re-design cycles before a design is fit for production. Chapter 2 has shown that feature-based design techniques (automatic feature recognition and design by features) represent a first step towards better manufacturability because they allow efficient mapping between geometric entities and manufacturing processes. However, geometric interactions between features alter a feature's properties and can generate undesired configurations that are impossible to produce with available production processes (2½D milling in the case of this thesis). Efficient validation tools are therefore needed by designers that help give early feedback on manufacturability issues related to proposed designs. These issues were presented in Chapter 3 with a particular focus on the manufacturability analysis of prismatic mechanical components.

In Chapter 4, software agent technology was introduced. It originates from the field of artificial intelligence and has emerged as a major new programming paradigm. It pushes forward the concepts of object-oriented programming by adding communication and autonomy to traditional objects. Agents are autonomous entities able to perceive and act on their environment in order to achieve their personal goals.

Chapter 5 has presented a new approach to the design-by-features approach, which uses autonomous software agents to embody geometric features. Feature agents are created that not only hold their geometric information but also contain embedded knowledge concerning manufacturability issues linked to the production processes used to create them. Using this concept of feature agents, it is possible to create a new type of CAD application that is trusted to assist the user in their design task. Indeed each feature agent is given a degree of control over its geometry and is allowed to modify itself. Features apply their embedded knowledge and problem solving capabilities to ensure their local manufacturability. Repetitive tasks such as checking for tool accessibility and thin sections can be delegated to feature agents equipped with adequate knowledge. In effect, the designer surrenders part of their non-critical duties to autonomous software entities, which enables them to concentrate on more crucial aspects of the design process.

An innovative architecture was proposed in Chapter 6 that uses agents driven by internal desires and motivated by dynamic beliefs. Individual features, implemented as agents, dynamically build a representation of their environment and use it to determine their local manufacturability. On detecting a range of pre-defined problems, agents apply their embedded problem solving knowledge to determine how they can modify themselves to achieve their goal (a machinable state). A working experimental implementation (MADSfm) of this architecture was presented that attests to the feasibility of the proposed agentification of features. The resulting CAD application allows creation of 2½D components using a limited library of features agents. It creates an original active product model, which represents the centre of activity for the system. This active model achieves real-time analysis of the design and performs autonomous geometric corrections to ensure its manufacturability. Moreover, a behaviour selection mechanism was presented in Chapter 6 that allows the user to specify their design intent in a novel fashion. Alternative behaviours can be selected for individual validation criteria adapting a feature's conduct to the designer's needs. The addition of service agents providing complex services to the proposed lightweight feature agents was also shown to provide a powerful extension mechanism to the CAD system.

The agent driven CAD system obtained offers a substantially different set of functions to the designer than more traditional packages. In particular, the system offers immediate design analysis and autonomous model corrections to ensure manufacturability of the design. However, the issue of livelock and the damaging hyperactivity they create remains partially unresolved.

8.2 Summary of conclusions

This thesis has presented a novel approach for ensuring better manufacturability of 2½D mechanical components. It uses the emerging agent technology in conjunction with a "design by features" approach to CAD thereby creating autonomous feature agents. Embedded knowledge about manufacturing capabilities is applied within the design environment to bring forward manufacturing considerations during the geometric modelling of components. A degree of geometric freedom is granted to features and the user trusts them to handle well-understood manufacturing issues such as tool access and thin sections. A global emergent function of the design system is to perform automatic analysis and autonomous correction of the designed component.

The various objectives set in Chapter 1 were completed and provided valuable material to support the broader research issues discussed in this thesis.

- Experiments were carried out with different level of agency. Purely reactive agents were tested in a simple 2D problem (see section 6.2.1.a). An original hybrid approach combining BDI and reactive behaviours was also tested (see section 6.2.1.b) that addresses the weaknesses of reactive agency. This motivated reaction scheme was adopted for implementing feature agents.

- The field of research addressing feature validity was surveyed and five validation rules (presence, proximity, access, collision and minimality) selected to perform manufacturability analysis. Basic solving strategies were devised that feature agents use to enforce these rules.

- The prototype system MADSfm was designed and implemented (see Chapter 6). It provides a 2½D modelling environment for mechanical design and proves able to perform analysis and self-modification with very short response time (see section 7.4).

- The prototype system implemented permits the creation of an active geometric model that provides continuous manufacturability analysis and autonomous geometric modification of features. Internal agent activity results in the active model assisting the user in handling manufacturability issues during design.

- A space-partitioning scheme based on octree encoding was created as a service agent inside the prototype system (see section 6.3.4.d). It greatly reduces the communication load within the system. The local operation of features does not

degrade the system's performance (see section 7.4.7), thereby demonstrating the principle of locality in feature models.

The presentation and implementation of this new approach has raised a number of important topics that deserve summarising.

8.2.1 Active product model

Agentifying individual features inside the system creates an active model that replaces the passive data structure used in conventional systems. A living community of autonomous software agents represents the component model. Within it, each feature agent autonomously performs analysis of its local geometry. When potential problems are detected, feature agents apply template solving-strategies and modify themselves to ensure local manufacturability. All activity inside the agent community results from autonomous inter-agent communication and no user intervention is required for the model to perform its tasks.

8.2.2 Architecture changes

The use of autonomous agents to embody design features involves major changes to the global CAD system architecture. These modifications, fully described in section 5.3.1 and 6.3.3, are summarised here:

- Parallel and distributed processing.

  Autonomous agents provide a natural scheme for parallel and distributed processing. Indeed, each autonomous software agent can perform its activity locally and relies only on peer-to-peer communication for sensing and acting on its environment. Agent technology has proven its applicability to solving distributed problems, which require decomposition into loosely dependent sub-tasks. Such sub-tasks (or elements) can be turned into autonomous agents and are prime candidates for execution on parallel/distributed computers.

- Peer-to-peer interaction:

  The traditional client/server interaction is replaced by a more flexible peer-to-peer scheme that allows any agent in the system to initiate dialog with any other agent. The peer-to-peer model eliminates the needs for central control and allows activity to be carried out locally. Agents use communication to propagate changes through the model and maintain the global activity of the system (see section 6.3.4.e). In such a

system, the user is also assimilated to an agent and can interact with other agents on a peer-to-peer basis.

- Time continuity:

  Unlike traditional systems, agent-based programs must perform their task over-time rather than punctually. Indeed, each feature must build up its local dynamic knowledge of its surrounding environment in order to perform its analysis and solving duties. This dynamic knowledge is accumulated over time through the inter-agent communication within the model. Therefore, autonomous agents require time continuity in order to realise their potential fully. In practice, this means that the agent community representing a model should be left *running* as long as practically feasible during design sessions. Also, a *learning* time lapse may be required for freshly loaded models before the system reaches a fully operational state.

8.2.3 Advantages for the designer

The active product model assists the designer during the design process. Indeed, the model is allowed to perform geometric self-corrections on the designer's behalf to solve template manufacturability problems. In particular, the model is able to autonomously handle some of the consequences, in terms of manufacturability, of changes performed by the designer. This delegation of tasks allows designers to concentrate on critical aspects of the design while feature-agents deals with less important and more repetitive duties.

8.2.4 Drawbacks for the designer

The geometric autonomy granted to features agents could create unstable models that inadvertently destroy part, or all, of the designed component. This potential instability can be avoided through the careful use of dynamic behaviour selection. However, it is felt that assisted (e.g. partially automated) behaviour selection is required to avoid adding a significant new burden to the designers. The unresolved issue of livelock could also create problems as it has potential destructive consequences on the model.

8.3 Further research

This thesis covers the basic concepts related to feature agents and their use to perform manufacturability testing. The prototype realised as a proof of concept falls short of providing a complete modelling environment and would require many improvements before

being usable in real design projects. A number of areas have been identified as offering attractive possibilities for further research and development.

8.3.1 Integration with a 3D geometric kernel

Various limitations in the computing environment used to realise the prototype system (MADSfm) prevented the feature agents using a full 3D geometric kernel. Instead, a limited 2½D library had to be created from scratch to support all geometric activity of the system. This implementation decision considerably limits the capabilities of the system. It is felt that using a proven industrial-strength kernel such as ACIS would greatly improve the system geometric abilities as well as its robustness. Recent advances in the development tools used (Java-Swarm and Scheme-Swarm in particular) make this integration a more feasible task.

8.3.2 Complex agent co-operation

MADSfm uses very basic interaction protocol to co-ordinate the activity of feature and service agents. The artificial intelligence community has researched more complex co-operation modes that could increase both functionality and robustness of the approach presented in this thesis. Delegation schemes such as contract-nets (see section 4.4.1) could bring enhanced capability to individual features. In particular, more complex solving technique could be used that require co-operation between multiple features.

8.3.3 Constraint as part of an agent's goal

A primitive implementation of inter-feature constraint management is included within MADSfm (see section 6.3.4.d and 7.3.2). However, it is only intended to show that traditional constraint-based modelling is not incompatible with autonomous agent activity inside the active model. It uses a service agent to track and enforce passive constraints between features. A more flexible and powerful way to support constraints between features would be to create dynamic desires within feature agents.

8.3.4 New validation rules and solving behaviours

The validation rules used by MADSfm cover a limited number of manufacturability issues related to 2½D geometry. Other validation criteria have been identified **[56]** and could be added to the existing system. The geometric validity of a feature is believed to be an important rule that should be added to the system. For example, validation rules could ensure

that features do not modify their aspect ratio beyond the capability of the machining process (see section 7.2.2.b). The addition of new solving behaviours for existing criteria would also enhance the flexibility of the system. Indeed, libraries of alternative behaviours would help to fit designer's particular needs.

8.3.5 Layered behaviours

The behaviour selection mechanism used in MADSfm provides limited flexibility and could greatly benefit from more advanced capabilities. It is thought that a subsumption scheme (see section 4.3.5.b) could help creating layered behaviour responses for feature-agents.

In MADSfm, agents use one independent solving routine for each supported manufacturability criteria. Although selected among several alternative behaviours, it does not offer much flexibility. Indeed, when the selected behaviour proves insufficient to solve a given problem, the feature has no option but to continue its activity in degraded mode until the problem is solved by the designer or another agent. In contrast, a layered behaviour scheme would allow each feature to be equipped with an ordered list of behaviours with increasing complexity. These behaviours could be applied in order of priority until detected problems are solved. High priority behaviours could be set that handle straightforward situations, in an efficient manner, using simple algorithms. Lower priority behaviours would use more complex algorithms and target convoluted situations. Simple algorithms could be tried first to eliminate detected problems. Unsuccessful applications of one algorithm would lead to the automatic selection of a more advanced solving behaviour until a solution is found or no other behaviour exists.

8.3.6 Agent learning

A degree of *learning* is already present in the system presented in this thesis. Indeed, it has been seen that agents dynamically build a view of their environment in order to make decisions (see section 5.3.4). However this learning process only involves collecting data from the surroundings. A more advanced form of learning capability would be to allow feature agents to dynamically modify their behavioural responses according to their accumulated experiences. Agents could analyse past activity to identify repeated patterns and generate new courses of actions based on them. The newly generated behavioural responses should be adapted more closely to the current situation since they are generated as a result of activity taking place inside the model.

8.3.7 Feature Pattern agents

In its current state, MADSfm only supports activity of individual geometric features. However, its flexible architecture should allow the creation of aggregated feature types such as feature patterns. Rectangular and circular patterns are not uncommon in mechanical design and could also benefit from the autonomous activity of agents.

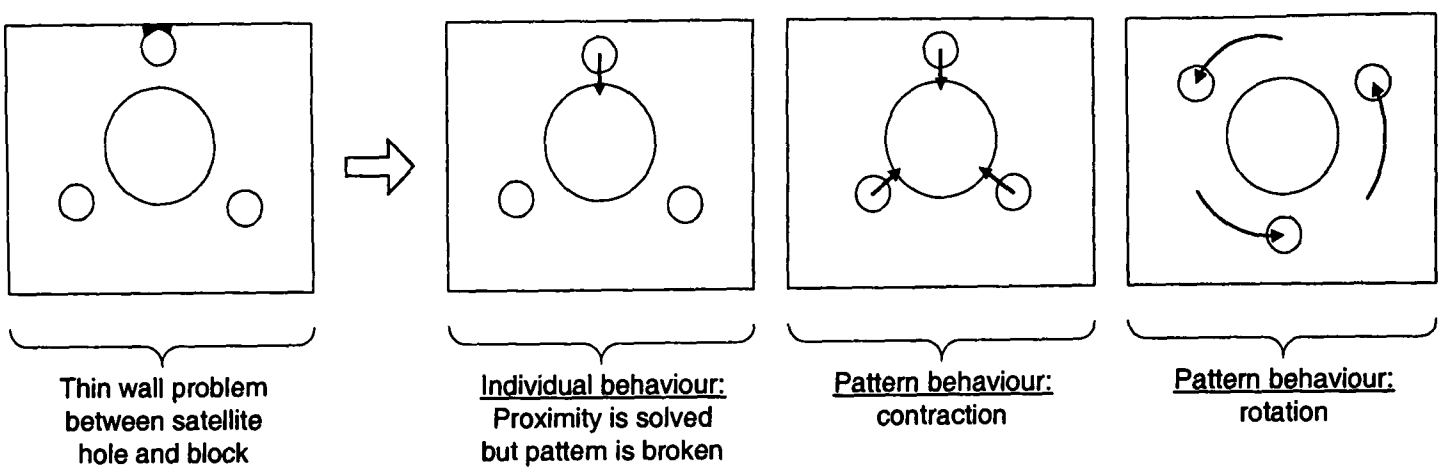


Figure 8-1: Example of useful pattern behaviours

Indeed, solving strategies applied by individual features might not be suitable for patterned featured. Moreover, the user would benefit greatly from specific pattern behaviours as shown in Figure 8-1.

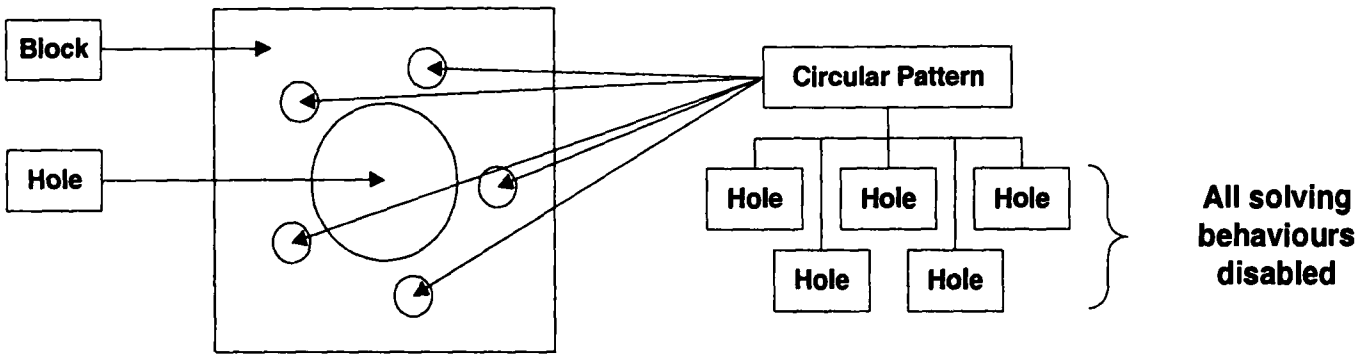


Figure 8-2: A possible scheme for Feature Pattern Agents

Figure 8-2 illustrates a possible way of bringing specific pattern behaviours to the system that takes advantage of the existing feature's expertise. A new pattern agent type could be created that provides pattern behaviours and uses the geometric knowledge of individual features. The pattern agent exists as a non-geometric agent and is linked to individual features whose solving behaviours have been disabled. This way the individual features still perform autonomous geometric analysis, but leaves the solving to the pattern agent instead of using their own solving routine.

8.3.8 Application to part assembly

This thesis has presented and demonstrated a system in which feature-agents can be trusted to take autonomous decisions and modify their geometry to achieve their goals. This autonomy principle has only been applied to manufacturability analysis but could be used in other areas of mechanical design. In particular, it is believed that feature-agents could provide valuable assistance in the creation of assemblies of feature-based components.

Equipped with adequate embedded knowledge related to assembly, features in different component could be made to co-operate within an assembly system in order to reach assemblable configurations.

8.3.9 Increased user/agent interaction

The peer-to-peer nature of agent interaction has been shown to be a fundamental element of their usefulness. Their ability to initiate dialog with each other, in particular, alleviate the limitations of the traditional client/server interaction model. It is believed that this initiative should be used fully to provide a higher degree of interaction between the designer and the system. The implementation realised with MADSfm did not fully utilise this capacity of features and user to engage in peer-to-peer interaction. Indeed, although the user was allowed to initiate dialog with any agent in the model on a peer-to-peer basis, the opposite was not permitted. Allowing features to autonomously query the user could make the system more interactive.

References

1. **PART white paper**
   Mitchell T. De Jong, Alex Fuchs, White paper, ICEM technologies, Minnesota, USA, 1994

2. **Design for machining with a simultaneous-engineering workstation**
   F. G. Mill, J. C. Naish and J. C. Salmon, Computer-aided design, Volume 26, Number 7, Pages 251-257, July 1994

3. **SIPS: An Application of Hierarchical Knowledge Clustering to Process Planning**
   D.S. Nau and M. Gray, In Symposium on Integrated and Intelligent Manufacturing at ASME Winter Annual Meeting, Pages 219-225. Anaheim, CA, December 1986

4. **HutCAPP A machining operations planner**
   Martti Mäntylä and Jussi Opas, In The Second International Symposium on Robotics and Manufacturing Research, New York, ASME Press, 1988

5. **CAPP, CAM-I automated process planning system**
   C.H. Link, Proceedings of 1976 NC Conference, CAM-I, Inc., Arlington, Texas, 1976

6. **Making sense out of agents**
   James Hendler, IEEE Intelligent Systems, Volume 14, Number 2, Pages 32-37, march/april 1999

7. **A roadmap of agent research and development**
   Nicholas R. Jennings, Katia Sycara and Michael Wooldridge, Autonomous Agents and Multiagent systems, Volume 1, Number 1, Pages 7-38, 1998

8. **Applications of intelligent agents**
   N.R. Jennings, M. Wooldridge, Agent Technology: foundations, applications and markets, Springer-Verlag, ISBN 3-540-63591-2, 1998

9. **Agents and software engineering**
Michael Wooldridge, AI*IA Notizie, Volume 6, Number 3, Pages 31-37, 1998

10. **Emergence-based Cooperation in a Multi-Agent System**
Fabrice Chantemargue, Thierry Dagaeff and Beat Hirsbrunner, Proceeding of the Second Conference on Cognitive Science (ECCS'97), Manchester, UK, Pages 91-96, April 9-11 1997

11. **MANTA: new experimental results on the emergence of (artificial) ant societies**
A. Drogoul, B. Corbara and D. Fresneau, in Proceedings of Simulating societies symposium, Siena , C. castelfranchi (Ed.), 1993

12. **Automated manufacturability analysis: a survey**
Satyandra K. Gupta, William C. Regli, Diganta Das and Dana S. Nau, Research in Engineering Design, Volume 9, Number 3, Pages 168-190, 1997

13. **Designing with features: the origin of features**
J.J. Cunningham and J.R. Dixon, ASME Computers in Engineering Conference, CIE, San Francisco, Pages 237-243, 1988

14. **Graph-based Extraction of Topological Features in Solid Models**
M. R. Henderson and P. S. Gavankar, Computer-Aided Design, Volume 22, Number 7, Pages 442-450, September 1990

15. **Parametric and feature-based CAD/CAM**
Jami J. Shah and Martti Mantyla, John Wiley & Sons, inc., ISBN 0-471-00214-3, 1995

16. **Solid modeling and beyond**
Aristide A.G. Requicha, IEEE Computer Graphics and Applications, Volume 12, Number 5, Pages 31-44, September 1992

17. **Automated Process Planning using Hierarchical Abstraction**
D.S. Nau, Texas Instrument Technical Journal, Pages 39-46, Winter 1987

18. **An industry-based study of cutting process capability representation requirements for an integrated simultaneous engineering workstation**
Jane C. Naish, Frank G. Mill and Jonathan C. Salmon, IIE transactions, Volume 29, Number 7, Pages 573-584, 1997

19. **Process capability modelling for manufacturing process selection in an integrated simultaneous engineering workstation**
Jane Naish, Ph.D. thesis, University of Strathclyde in Glasgow, 1998

20. **3D geometric reasoning algorithms for feature recognition**
JungHyun Han, PhD computer science, University of Southern California, august 1996

21. **Machine Interpretation of CAD Data for Manufacturing Applications**
Qiang Ji and Michael M. Marefat, ACM Computing Surveys, Volume 24, Number 3, Pages 264-311, September 1997

22. **A survey of automated feature recognition techniques**
W.C. Regli, Technical Report TR 92-18, University of Maryland, College Park, MD 20742, 1992

23. **Shape classification in computer-aided design**
L.K. Kyprianou, Ph.D. thesis, Christ's college, University of Cambridge, 1980

24. **Using automatic feature recognition to interface CAD and CAPP**
E. Wang, in Proceedings of ASME Conference on Computers in Engineering, Volume 1, Pages 215-231, San Francisco, California, August 1992

25. **Method for finding holes and pockets that connect multiple faces in 2 ½D object**
J. Corney and D.E.R. Clarck, Computer-aided design, Volume 23, Number 10, Pages 658-668, 1991

26. **The Heriot-Watt feature finder: A graph-based approach to recognition**
G. Little, R. Tuttle, D.E.R. Clark, J. Corney, ASME-CIE 1997, Sacramento, Pages 14-17, September 1997

27. **Extracting and identifying form features: a Bayesian approach**
Michael M. Marefat and Qiang Ji, IEEE International Conference On Robotics and Automation, San Diego, CA, USA, Pages 1959-1964, 1994

28. **Coupling rule-based and object-oriented programming for the classification of machined features**
K.E. Hummel, ASME Computer in Engineering Conference, Anaheim, CA, USA, Pages 409-418, 1989

29. **Feature extraction by volume decomposition**
T. Woo, in Proceedings conference CAD/CAM technology in mechanical engineering, MIT Cambridge, USA, Pages 76-94, 1982

30. **Convex hull-based feature-recognition method for 2.5D components**
J.C.E. Ferreira and S. Hinduja, Computer-aided design, Volume 22, Number 1, Pages 41-49, January/February 1990

31. **Form feature recognition using convex decomposition: results presented at the 1997 ASME feature panel session**
E. Wang and Y.S. Kim, Computer-aided design, Volume 30, Number 13, Pages 983-990, November 1998

32. **Recognition of form features using convex decomposition**
Y.S. Kim, Computer-aided design, Volume 24, Number 9, Pages 461-477, 1992

33. **Form feature recognition using convex decomposition: Geometric domain extension**
Y. S. Kim, Proceeding of NSF design and manufacturing systems conference, La Jolla, CA, USA, Pages 121-122, ISBN 0-87263-455-2, January 1993

34. **Recognizing multiple interpretations of interacting machining features**
Y.J. Tseng, S.B. Joshi, Computer-aided design, Volume 26, Number 9, Pages 667-688, 1994

35. **Spatial Reasoning for the Automatic Recognition of machinable features in solid models**
Jan H. Vandenbrande, Aristide A. G. Requicha, IEEE Transaction on pattern analysis and machine intelligence, Volume 15, Number 12, Pages 1269-1285, December 1993

36. **Hint-based reasonning for feature recognition: status report**
J.H. Han, W.C. Regli and S. Brooks, Computer-aided design, Volume 30, Number 13, Pages 1003-1008, November 1998

37. **3D Modeling with the ACIS Kernel and toolkit.**
Jonathan Corney, John Wiley & Sons Inc., ISBN: 471-96535-9, 1997

38. **Extreme modeling handle the tough stuff**
Special edition of Machine design, Parasolid – Unigraphics solutions, 1999
http://www.parasolid.com

39. **What is a Parametric Family of Solids?**
V. Shapiro and D.L. Vossler, Proceedings of the 3rd ACM/IEEE Symposium on Solid Modeling and Applications, ACM Press, Salt Lake City, Utah, May 17-19, 1995

40. **STEP an introduction (2nd edition)**
Jon Owen, Pub. Information Geometers, ISBN 1-874728-11-9, 1997

41. **A hierarchical structure for form features**
N.N.Z. Gindy, International Journal for Production Research, Volume 27, Number 12, Pages 2089-2103, December 1989

42. **Requirements for support of form features in a solid modeling system**
M.J. Pratt, P.R. and Wilson, Technical Report R-86-ASPP-01, CAM-I Inc., Arlington, Texas, 1986

43. **Part features for process planning**
W. Butterfield, M. Green and D. Stoker, Technical Report R-85-PPP-03, CAM-I Inc., Arlington, Texas, 1985

44. **Feature modelling and conversion - Key concepts to concurrent engineering**
Willem F. Bronsvoort and Frederik W. Jasen, Computers in Industry, Volume 21, Number 1, Pages 61-85, 1993

45. **Incremental feature modelling: methodology for integrating features and solid models**
Timo Laakko, Ph.D. Thesis, Helsinki University of Technology, 1993

46. **Object-Oriented analysis and design of a manufacturing feature representation**
A. Chep and L. Tricarico, International Journal for Production Research, Volume 37, Number 10, Pages 2349-2376, October 1999

47. **Feature attributes and their role in product modeling**
Somashekar Subrahmanyam, in Proceeddings of Solid Modeling '95, Salt Lake City, Utah, USA, ACM Press, Pages 115-124, 1995

48. **Features and the principle of locality in process planning**
J. Vancza, International Journal Computer Integrated Manufacturing, Volume 6, Number 12, Pages 126-136, January/April 1993

49. **User-defined features in EXTDesign++**
Timo Laakko, Martti Mantyla and Jussi Opas, Technical Report HTKK-TKO-B129, Helsinki University of Technology, Laboratory of Information Processing Science, December 1995

50. **Declarative user-defined feature classes**
Rafael Bidarra, Abdelfettah Idri, Alex Noort and Willem F. Bronsvoort, in CD-ROM Proceedings of DETC'98 ASME Design Engineering Technical Conference, Atlanta, 1998

51. **On user-defined features**
C.M. Hoffmann and R. Joan-Aringo, Computer-aided design, Volume 30, Number 5, Pages 321-332, April 98

52. **Interactive feature definition**
Salomons O.W., Slooten F. van, Jonker H.G., Houten F.J.A.M. van and Kals H.J.J., Int. Conf. On Feature modelling and recognition in advanced CAD/CAM systems, Valenciennes, France, Pages 181-200, 1994

53. **Handling feature interactions in concurrent design and manufacturing**
R.R. Karinthi, D.S. Nau and Q. Yang, Applied Artificial Intelligence, Volume 6, Number 4, Pages 389-415, 1992

54. **Representing and manipulating interacting and inter-feature relationships in engineering design for manufacture**
R.E. Da Silva, K.L. Wood and J.J. Beaman, in Proceedings conference ASME design automation conference, Chicago, Ill, USA, Pages 1-8, September 1990

55. **What are feature interactions?**
William C. Regli and Michael J. Pratt, in CD-ROM proceedings of ASME 1996 DETC and CIE, Irvine, CA, august 1996

56. **Automatic Detection of Interactions in Feature Models**
Rafael Bidarra, Maurice Dohmen and Willem F. Bronsvoort, Proceedings of ASME Design Engineering Technical Conferences, Sacramento, California, Pages 14-17 September 1997

57. **What is a manufacturing interaction?**
W. Faheem, C.C. Hayes, J.F. Castano and D.M. Gaines, in CD-ROM Proceedings of ASME DETC98 DFM, Atlanta, Georgia, September 1998

58. **Feature-based design by volumetric machining features**
T.N. Wong and K.W. Wong, International Journal for Production Research, Volume 36, Number 10, Pages 2839-2862, October 1998

59. **Feature-based interaction: an identification and classification methodology**
M. Da Silva Hounsell and K. Case, IMechE Proc Insts Mech Engr Part B: Journal of Engineering Manufacture, Volume 213, Number B4, Pages 369-380, ISNN 0954-4054369-380, 1999

60. **The Edinburgh composite component**
P. Husband, F. Mill, G. Pedley and S. Warrington, in Proceedings of 5th international conference On manufacturing science and technology of the future, Enschede, June 1991

61. **Representation problems in feature-based approaches to design and process planning**
F. G. Mill, J. C. Salmon and A. G. Pedley, Int. J. Computer Integrated Manufacturing, Volume 6, Number 1, Pages 27-33, 1993

62. **Geometric reasoning for process planning**
J.C. Salmon, Ph.D. thesis, School of Mechanical Engineering, University of Edinburgh, 1997

63. **Validity maintenance of Models with interacting features**
Rafael Bidarra and Willem F. Bronsvoort, Proceedings of the Eighth Portuguese Conference on Computer Graphics, Coimbra, Portugal, Pages 65-79, 1998

64. **Validity Maintenance of Semantic Feature Models**
Rafael Bidarra and Willem F. Bronsvoort, Proceedings of Solid Modeling '99 - Fifth Symposium on Solid Modeling and Applications, ACM Press, Pages 85-96, 1999

65. **Validity maintenance in semantic feature modelling**
R. Bidara, Ph.D. thesis, Delft University of Technology, Universal Press, ISBN 90-9012599-X, 1999

66. **Automated feature validation for creating/editing feature-based component data models**
N.N.Z. Gindy, Y. Yue and C.F. Zhu, International Journal for Production Research, Volume 36, Number 9, Pages 2479-2495, September 1998

67. **Feature Validation in a Multiple-View Modeling System**
Maurice Dohmen, Klass Jan de Kraker and Willem F. Bronsvoort, in CD-ROM proceedings of ASME CIE 96, Irvine, California, August 1996

68. **Issues on feature-based editing and interrogation of solid models**
Jaroslaw R. Rossignac, Computers & Graphics, Volume 14, Number 2, Pages 149-172, 1990

69. **Modeling with self validation features**
Ferruccio Mandorli, Umberto Cigini, Haralf E. Otto and Fumihiko Kimura, in Proceedings of ACM/IEEE symposium on Solid Modeling and Applications '97, Atlanta, ACM Press, Pages 88-96, 1997

70. **History-independent boundary evaluation for feature modeling**
Rafael Bidarra and Willem F. Bronsvoort, in CD-ROM Proceedings of DETC'98 1999 ASME Design Engineering Technical Conferences, Las Vegas, September 1999

71. **Feature transformations between application-specific feature spaces**
Jami J. Shah, Computer-Aided Engineering Journal, Pages 247-255, December 1988

72. **Feature mapping and application shell**
J.J. Shah, A. Bhatnagar and D. Hsiao, in Proceedings of ASME Computers in Engineering Conference (CIE), San Francisco, Pages 489-496, 1988

73. **A feature-based design system and its potential to unify CAD and CAM**
H. Denzel and G.C. Vosniakos, Proceedings of IFIP workshop in interfaces in industrial systems for production and engineering, Germany, March 15-17, IFIP transactions B: Applications in Technology, Pages 131-144, 1993

74. **Feature-based modelling by integrating design and recognition approaches**
De Martino T., Falcidieno B., Giannini F., Hassinger S., Ovtcharova J., Computer-Aided Design, Volume 26, Number 8, Pages 646-653, August 1994

75. **Design and engineering process integration through multiple view intermediate modeller in a distributed object-oriented system environment**
T. De Martino, B. Falcidieno and S. Habinger, Computer-Aided Engineering, Volume 30, Number 6, Pages 437-452, May1998

76. **Feature-based modelling of product families**
T. Laakko and M. Mantyla, The 1994 ASME International Computers in Engineering Conference, Volume 1, Pages 45-54, New-York, September 1994

77. **Incremental constraint modelling**
Timo Laakko and Martti Mantyla, Chapter 19 of "Advances in Feature-Based Manufacturing", Elsevier science pub, Amsterdam, Pages 455-479, 1994

78. **Manufacturability Evaluation and Generation of Re-Design Suggestions for Machined Parts**
S.K. Ong and A.Y.C. Nee, The International Journal for Manufacturing Science & Production, Volume 1, Number 2, Pages 87-105, 1998

79. **Measuring Geometric Complexity of 3D Models for Feature Recognition**
G.Little,R. Tuttle, D.E.R. Clark and J.R. Corney, International Journal of Shape

Modelling, Volume 3, Number 3 & 4, Pages 141-154, Pub World Scientific Publishing Company, ISSN 0218-6543, September and December 1997

80. **A Feature Complexity Index**
G. Little, R. Tuttle, J.R. Corney and D.E.R. Clark, IMechE Journal of Mechanical Engineering Science - Part C. Volume 212, Number C5, Pages 405-413, ISSN 0954-4062, 1998

81. **Manufacturing Intelligence**
Paul Kenneth Wright and David Alan Bourne, 1988, Addison-Wesley Publishing Company Inc., ISBN 0-201-13576-0

82. **Fixture Planning in a Feature Based Environment**
Ser Chong Chia, Ph.D. thesis, School of Mechanical Engineering, University of Edinburgh, 1997

83. **Computer Aided Manufacturing**
Tien-Chien Chang, Richard A. Wysk and Hsu-Pin Wang, Prentice-Hall Inc., ISBN 0-13-161571-8, 1991

84. **CyberCut: A Networked Manufacturing Services**
P. K. Wright and D.A. Dornfield, Transactions of NAMRC/SME, Volume XXVI, 1998

85. **Cutting faster, thinner, and bigger**
Chris Koepfer, Modern Machine Shop Online, http://www.mmsonline.com/articles/089709.htm, 1997

86. **A systematic approach for analysing the manufacturability of machined parts**
Satyandra K. Gupta and Dana S. Nau, Computer Aided Design, Volume 27, Number 5, Pages 343-352, 1995

87. **Knowledge-based manufacturability assessment: an object-oriented approach**
Yuh-min Chen, Allen Miller and Korhan Sevenler, Journal of Intelligent Manufacturing, Volume 6, Pages 321-337, 1995

88. **Generating redesign suggestions to reduce setup cost: a step towards automated redesign**
Diganta Das, Satyandra K. Gupta and Dana S. Nau, Computer-Aided design, Volume 28, Number 10, Pages 763-782, 1996

89. **Working with multiple representations in a concurrent design system**
M.R. Cutkosky, J.M. Tenenbaum and D.R. Brown, ASME Journal of Mechanical Design, Volume 114, Number 3, Pages 515-524, 1992

90. **Features in Process Based Desigu**
M.R. Cutkosky, J.M. Tenenbaum and D. Muller, Proceedings of the ASME International Computers in Engineering Conference and Exhibition, July 31-August 4, 1988

91. **Feature Recognition for Manufacturability Analysis**
W.C. Regli, S.K. Gupta and D.S. Nau, in Proceeding of ASME-CIE 1994, Pages 93-104, September 1994

92. **Automated Manufacturability Analysis of Machined Parts**
Satyandra K. Gupta, PhD. Thesis, University of Maryland, USA, 1994

93. **Integrating DFM with CAD through Design Critiquing**
S.K. Gupta, W.C. Regli and D.S. Nau, Concurrent Engineering: Research and Applications, Volume 2, Number 2, Pages 85-95, 1994

94. **Computer support in the design of mechanical products**
Otto Willem Salomons, Ph.D. thesis, University of Twente, Netherland, 1995
http://www.pt.wb.utwente.nl/staff/otto/thesis

95. **An introduction to agent technology**
H. S. Nwana and D. T. Ndumu, in Software agents and Soft Computing, Lecture Notes in Artificial Intelligence LNAI 1198, Springer-Verlag, ISBN 3-540-62560-7, Pages 5-26, 1997

96. **Agent SourceBook**
Alper Caglayan and Colin Harrison, Wiley & sons, Inc., ISBN 0-471-15327-3, 1997

97. **Is it an agent, or just a program?: A taxonomy for autonomous agents**
Stan Franklin and Art Graesser, ECAI 96 Workshop (ATAL), Budapest, Hungary, August 1996
also in Intelligent Agents III, Lecture Notes in Artificial Intelligence LNAI 1193, Springer-Verlag, ISBN 3-540-62507-0, 1996

98. **Multi-agent systems : an introduction to distributed artificial intelligence**
Jacques Ferber, Addison-Wesley, ISBN 0-201-36048-9, 1999

99. **Agent Theories, Architectures, and Languages: A Bibliography**
Michael Wooldridge, Jorp P. Muller and Milind Tambe,. ECAI 95 Workshop (ATAL),

Montreal, Quebec, August 1995
also in Intelligent Agents II, Lecture Notes in Artificial Intelligence LNAI 1037, Springer-Verlag, ISBN 3-540-60805-2, 1995

100. **Intelligent agent: Theory and Practice**
Michael Wooldridge and Nicholas R. Jennings, Knowledge Engineering Review, Volume 12, Number 2, Pages 115-152, 1995

101. **What's an agent, anyway? A sociological case study**
Leonard N. Foner, Agent Group, Technical Report, MIT Media Lab, E15-305, 20 Ames st, Cambridge, MA 02139, 1993

102. **Software agents**
Michael R. Genesereth and Steven P. Ketchpel, Communication of the ACM, Volume 37, Number 7, Pages 48-53, July 1994

103. **Evaluation of KQML as an Agent Communication Language**
James Mayfield, Yannis Labrou and Tim Finin, ,. ECAI 95 Workshop (ATAL)
Montreal, Quebec, August 1995
also in Intelligent Agents II, Lecture Notes in Artificial Intelligence LNAI 1037, Springer-Verlag, ISBN 3-540-60805-2, 1995,
http://www.cs.umbc.edu/~finin/finin-papers.shtml

104. **Go to the ants**
Van Dike Parunak, Annals of Operations Research, Volume 75, Pages 69-101, 1997, http://www.iti.org/~van/gotoant.ps

105. **Swarm intelligence, from natural to artificial systems**
Eric Bonobeau, Marco Dorico and Guy Theraulaz, Oxford University Press, ISBN 0-19-513159-2, 1999

106. **Multi-agent simulation as a tool for modelling societies: application to social differenciation in ant colonies**
A. Drogoul and J. Ferber, Artificial social systems, Volume 830, Pages 3-23, C. castelfranchi and E. Werner (Ed.), Springer-Verlag, 1994

107. **Coordination without communication: Experimental validation of focal point techniques**
Maier Fenster, Sarit Kraus, Jeffrey S. Rosenschein, ICMAS 95. Proceedings of the 1st International Conferance On Multi-agent systems, Menlo Park, CA, USA, Pages 102-108, 1995

108. **Touring Machines: An architecture for dynamic, rational, mobile agents**

Innes Andrew Gergusson, Ph.D. thesis, University of Cambridge, UK, October 1992

109. **Logical foundations of artificial intelligence**

M. R. Genesereth and N. Nilsson, Morgan Kaufmann Publishers: San mateo, CA, 1987

110. **What ants cannot do**

Eric Werner, 6th European workshop on modelling autonomous agents in a multi-agent world, MAAMAW'94, Odense, Denmark, August 1994
also in Distributed software agents and applications, Lecture Notes in Artificial Intelligence LNAI 1069, Springer-Verlag, ISBN 3-540-61157-6, 1994

111. **Flocks, Herds, and Shoals: A Distributed Behavioral Model**

Graig W. Reynolds, IEEE Computer Graphics, Volume 21, Number 4, Pages 25-34, July 1987

112. **Not bumping into things**

Graig W. Reynolds, notes for the SIGGRAPH '88 course Developments in Physically-Based Modeling, Atlanta, August 1988

113. **Steering behaviours for autonomous characters**

Graig W. Reynolds, in on-line Proceedings of Games Developer Conference, http://www.gdconf.com, San Jose, CA, ISA, March 1999

114. **ICHTIUS: architecture d'un systeme multiagents pour l'etude de structures agregatives**

R. Mesle, Rapport de DEA, LAFORIA Université Paris 6, 1994

115. **Robot Motion Planning: A Distributed Representation Approach.**

J. Barraquand and J.C. Latombe, International Journal of Robotics Research, Volume 10, Number 6, Pages 628-649, 1991

116. **Numerical Potential Field Techniques for Robot Path Planning**

J. Barraquand, B. Langlois, and J.C. Latombe., IEEE Transactions on Systems, Man, and Cybernetics, Volume 22, Number 2, Pages 224-241, 1992

117. **Reactive-system approaches to agent architectures**

Lee Jacho, 5th International Workshop (ATAL), Orlando, USA, July 1999
also in Intelligent Agent VI, Lecture Notes in Artificial Intelligence LNAI 1757, Springer-Verlag, ISBN 3-540-67200-1, 1999

118. **Cooperation between distributed agents through self-organisation**

Luc Steels, Proceedings of the 1st European Workshop on autonomous agents, Elsevier, North Holland, Pages 175-196, 1990

119. **Belief Revision in Multi-agent Systems**

Benedita Malheiro, N. R. Jennings and Eugenio Oliviera, ECAI 94. 11th European Conference on Artificial Intelligence, John Wiley & Sons, inc., Pages 294-298, 1994

120. **Reactive reasonning and planning**

Michael P. Georgeff and Amy L. Lansky, AAAI 87. Proceedings of the 6th National Conference on Artificial Intelligence, Seattle, WA, USA, Pages 677-692, 1987

121. **Multi-Agent Systems and Agent-Based Simulation**

Sichman, Conte and Gilbert (eds), LNAI series, volume 1534, Berlin, Springer-Verlag, December 1998

122. **Integrating User Interface Agents with Conventional Applications**

Henry Lieberman, Proceedings of the ACM Conference on Intelligent User Interfaces, San Francisco, CA, January 1998

123. **PARAgente: exploring the issues in agent-based user interfaces**

J. Alfredo Sanchez, Flavio S. Azevedo and John J. Leggett, Proceedings of the first International Conference On multiagent systems - ICMAS'95, San Francisco, CA, Pages 320-327, June 1995

124. **Design for manufacturability via agent interaction**

H. Robert Frost and Mark R. Cutkosky, in CD-ROM proceedings of ASME 1996. Design for Manufacturability Conference, Irvine, CA, USA, August 1996

125. **Agent support for design manufacturability**

Krishna N. Jha, Gary Coen, Andrea Morris, Ed Mytych and Judith Spering, in CD-ROM proceedings of ASME DETC98 DFM, Atlanta, Georgia, September 1998

126. **Agent support for design of aircraft parts**

Krishna N. Jha, Andrea Morris, Ed Mytych and Judith Spering, in CD-ROM proceedings of ASME DETC98 DFM, Atlanta, Georgia, September 1998

127. **Mobile Database Nodes for Manufacturing Information Management: a STEP based approach**

F. Chaxel, E. Bajic and J. Richard, Int. Journal Advanced Manufacturing Technologies, Volume 13, Pages 125-133, 1997

128. **Motion planning for an articulated robot: a multi-agent approach**

L. Overgaard, H.G. Petersen and J.W. Perram, Proceedings of MAAMAW'94, Springer-verlag, 1994

129. **A robust layered control system for mobile robots**

Rodney A. Brooks, MIT AI Lab Memo 864, September 1985

130. **The design of intelligent agents: A layered approach**

Jorg P. Muller, Springer, Lecture Note in Artificial Intelligence LNAI 1177, ISBN 3-540-62003-6, 1996

131. **Blackboard Systems II: Blackboard Application Systems, Blackboard Systems from a Knowledge Engineering Perspective.**

Penny Nii, AI Magazine, Volume 7, Number 3, Pages 82-106, 1986

132. **Blackboard Systems: The Blackboard Model of Problem Solving and the Evolution of Blackboard Architectures**

Penny Nii, AI Magazine, Volume 7, Number 2, Pages 38-64, 1986

133. **The Evolution of Blackboard Control Architectures**

N. Carver and V. Lesser, Expert Systems with Applications, Special Issue on The Blackboard Paradigm and Its Applications, Volume 7, Number 1, Pages 1-30, 1994

134. **The neural basis of behavioural choice in an artificial insect**

R.D. Beer and H.J. Chiel, From Animals to Animats: 1st conference on simulation of adaptive behaviours, MIT press, Cambridge, Pages 247-254, September 1990

135. **From tom-thumb to dockers: experiment with foraging robots**

A. Drogoul and J. Ferber, From Animals to Animats: 2nd conference on simulation of adaptive behaviours, Hawaii, MIT press, Cambridge, Pages 451-459, December 1992

136. **A pragmatic BDI Architecture**

Klaus Fisher, jorg P. Muller and Markus Pishel, ECAI 95 Workshop (ATAL), Montreal, Quebec, August 1995

also in Intelligent Agents II, Lecture Notes in Artificial Intelligence LNAI 1037, Springer-Verlag, ISBN 3-540-60805-2, 1995

137. **BDI Agents: from Theory to Practice**

Anand S. Rao and Michael P. Goergeff, Proceedings of the First International Conference on Multi-Agent Systems (ICMAS'95), San-Francisco, USA, June 1995

138. **Modelling and Design of Multi-Agent Systems**

David Kinny and Michael Georgeff, ECAI 96 Workshop (ATAL), Budapest, Hungary, August 1996

also in Intelligent Agents III, Lecture Notes in Artificial Intelligence LNAI 1193, Springer-Verlag, ISBN 3-540-62507-0, 1996

139. **Behavior of Agents Based on Mental States**

Takashi Katoh, Hideki Hara, Tetsuo Kinoshita, Kenji Sugawara and Norio Shiratori Proceedings of the 13th International Conference on Information Networking (ICOIN '98), Koganei, Japan, ISBN 0-8186-7225-0, Pages 199-204, 1998

140. **JAM: a BDI-theoretic Mobile Agent Architecture**

Marcus J. Huber, Proceedings of the Third International Conference on Autonomous Agents, 1999

141. **Systemes de pilotage auto-organises et gammes distribuees: methodes de conception et application a une machine-outil**

Vincent Patriti, Ph.D. thesis, CRAN - Université Nancy 1, 1998

142. **Practical reasoning with Procedural Knowledge**

Michael Wooldridge, Proceedings of the International Conference on Formal and Applied Practical Reasoning, June 1996

143. **Agent Technology Green Paper**

Agent Working Group, OMG Document ec/99-03-11, the object management group (OMG) 1999

http://www.objs.com/agility

144. **KQML - a language and protocol for knowledge and information exchange**

Tim Finin and Rich Fritzson, Technical Report CS 94-02, Computer Science department, University of Maryland, UMBC, Baltimore, 1994

145. **KQML as an agent communication language**

Tim Finnin, Yannis Labrou and James Mayfield, invited chapter in Jeff Bradshaw (Ed.), in "Software Agents", MIT Press, Cambridge, 1997

146. **A proposal for a new KQML specification**

Yannins Labrou and Tim Finin, Technical Report TR CS-97-03, Computer Science and Electrical Engineering Department, University of Maryland, February 1997

147. **Communicative actions for artificial agents**

Philip R. Cohen and Hector J. Levesque, Proceedings of the International Conference on Multi-Agent Systems, San-Francisco, CA, June 1995

148. **AgenTalk: Coordination Protocol Description for Multiagent Systems**

K. Kuwabara, T. Ishida, and N. Osato, in Proceedings of First International Conference on Multi-Agent Systems (ICMAS '95), Menlo Park, CA, USA, June 1995

149. **Pitfalls of Agent-Oriented Development**

Michael Wooldridge and Nicholas R. Jennings, Agents'98: proceedings of the 2nd International Conference on Autonomous Agents, ACM Press, May 1998

150. **Issues in Multiagent Design Systems**

Susan E. Lander, IEEE Expert, Volume 112, Number 2, Pages 18-26, march-april 1997

151. **Using single function agent to investigate conflict**

B. V. Dunskus, D. L. Grecu, D. C. Brown and I. Berker, Artificial Intelligence for Engineering Design, Analysis and Manufacturing, Volume 9, Pages 299-312, 1995

152. **Detecting and resolving conflicts among cooperating human and machine-based design agents**

Mark Klein, Artificial Intelligence for Engineering, Elsevier science Publishers, Volume 7, Pages 93-104, 1992

153. **A single function agent framework for task decomposition and conflict negociation**

R.T. Sreeram and P.K. Chawdhry, in CD-ROM proceedings of ASME DETC98 DFM, Atlanta, Georgia, September 1998

154. **Co-ordination in multi-agent systems**

H. S. Nwana, L. Lee and N. R. Jennings, in Software agents and soft computing, Lecture Notes in Artificial Intelligence LNAI 1198, Springer-Verlag, ISBN 3-540-62560-7, 1997

155. **JATLite: A Java Agent Infrastructure with Message Routing**

H. Jeon, C. Petrie and M. R. Cutkosky, IEEE Internet Computing, Volume 14, Number 2, March/April 2000.

http://java.stanford.edu

156. **The Swarm Simulation System: A Toolkit for Building Multi-Agent Simulations**

Nelson Minar, Roger Burkhart, Chris Langton and Manor Askenazi, Technical Report,

SantaFe Institute, Santa-fe, CA, 1996
http://www.swarm.org

157. **COALA: Un Langage pour la Conception de Systèmes Adaptatifs de Résolution de Problèmes.**
O. Baujard, C. Spinu, C. Garbay, S. Pesty and J.M. Chassery, Colloque langage a objets, Grenoble 1994.

158. **The Common Object Request Broker: Architecture and Specification**
the object management group (OMG), 1995-1999
http://www.omg.org

159. **DCOM architecture white paper**
microsoft corporation, 1998
http://www.microsoft.com

160. **Software agent technologies**
H. S. Nwana and Michael Wooldridge, in Software agents and soft computing, Lecture Notes in Artificial Intelligence LNAI 1198, Springer-Verlag, ISBN 3-540-62560-7, 1997

161. **Knowledge Interchange Format Version 3, Reference Manual**
M.G. Genesereth and R.E. Fikes, Technical Report, Computer Science Department, Stanford University, 1992

162. **The Ontolingua Server: A Tool for Collaborative Ontology Construction.**
A. Farquhar, R. Fikes, and J. Rice, Technical report, Knowledge Systems Laboratory, Standford University, 1996.

163. **The Open Agent Architecture: A framework for building distributed software systems**
D. Martin, A. Cheyer and D. Moran, Applied artificial Intelligence: An International Journal, Volume 3, Number 1, Pages 91-128, January 1999

164. **Agents for the Masses**
Thompson, Bannon, Pazandak, and Vasudevan, invited paper, in proceedings of Agent 99 Workshop on Agent-Based High Performance Computing: Problem Solving Applications And Practical Deployment, Seattle, May 1 1999.

165. **FIPA specifications**
Foundation for Intelligent Physical Agents, Geneva, Switzerland, 1997-1998
http://drogo.cselt.stet.it/fipa

166. **KAoS: an open agent architecture supporting reuse, interoperability, and extensibility**
Jeffrey M. Bradshaw, in proceedings of Tenth Knowledge Acquisition for Knowledge-Based Systems Workshop, 1996

167. **Primitive interaction protocols for agents in a dynamic environment**
Roberto A. Flores and Niek Wijngaards, Twelfth Workshop on Knowledge Acquisition, Modeling and Management, Banff, Alberta, Canada, October 1999

168. **CRAASH: a coordinated collision avoidance system**
K. Zeghal, J. Ferber, in proceedings of European simulation multi-conference, Lyon, France, Pages 406-410, 1993

169. **An agent-based approach to concurrent cable-harness design**
Hisup Park, mark R. Cutkosky, Andrew B. Conru and Soo-hong Lee, Artificial Intelligence for Engineering Design, Analysis and Manufacturing., Volume 8, Pages 45-61, March 1994

170. **Next-Link: An experiment in coordination of distributed agents**
C. Petrie, M. Cutkosky, T. Webster, A. Conru and H. Park, Position paper in AID-94 Workshop on Conflict Resolution, Lausanne, 1994

171. **Agent-Based Collaborative Design of Parts in Assembly**
Toshiki Mori and Mark R. Cutkosky, in CD-ROM Proceeding of DETC'98 - ASME Design Engineering Technical Conferences, Atlanta, Georgia, USA, September 1998

172. **The AARIA agent architecture : an example of requirement-driven agent-based system design**
H.V.D. Parunak, A.D. Baker and S.J. Clark, ICAA'97, Proceeding of the First International Conference on Autonomous Agents, Marina del Rey, CA, USA, ACM Press, February 6-8, 1997
http://www.aaria.uc.edu

173. **The AARIA agent architecture: from manufacturing requirement to agent-based system design**
Van Dyke Parunak, Abert D. Baker and Steven J. Clark, Worshop on agent-based manufacturing, ICAA'98, Minneapolis, MN, May 1998

174. **Coordinating Societies of Research Agents - IMS Experience**
Brian R. Gaines and Douglas H. Norrie
Integrated Computer-Aided Engineering, Volume 4, Number 3, Pages 179-190, 1997

175. **A multi-agent intelligent design system integrating manufacturing and floor-shop control**

Sivaram Balasubramanian and Douglas H. Norrie, ICMAS 95. Proceedings of the 1st International Conference On Multi-agent systems, Menlo Park, CA, USA, Pages 3-9, June 1995

176. **A multiagent architecture for Concurrent Design, Process Planning, Routing and Scheduling**

S. Balasubramanian and D. H. Norrie, Concurrent Engineering: Research and Applications, Volume 4, Number 1, Pages 7-16, march 1996

177. **Software Agents**

Jeffrey M. Bradshaw, AAAI press and MIT press, ISBN 0-262-52234-9, 1997

178. **Features, aka the semantics of a formal language of manufacturing**

K.N. Brown, C.A. McMahon and J.H. Sims Williams, Research in Engineering Design, Volume 7, Pages 151-172, 1995

179. **Multi-agent systems and manufacturing**

Jean-Pierre Muller and H. Van Dyke Parunak, IFAC - INCOM'98, Nancy & Metz – France, Volume I, Pages 165-170, June 1998

180. **Massively parallel computing: status and prospects**

D.J. Walace, Technical report 9202, EPCC University of Edinburgh, 1992

181. **MultiProcessor Specification** (version 1.4)

Intel Corporation, 1997

http://www.intel.com

182. **Beowulf: a parallel workstation for scientific computation**

Donald J. Becker, Thomas Sterling, Daniel Savarese, John E. Dorband, Udaya A. Ranawak and Charles V. Packer, Proceedings, International Conference on Parallel Processing, 95, 1995

http://www.beowulf.org

183. **Parallel, Distributed and multiagent production systems**

Toru Ishida, Lecture Notes in Artificial Intelligence LANI 878, Springer-Verlag, ISBN 3-540-58698-9, 1994

184. **The application of multiagent systems to concurrent engineering**

David.C. Brown, Susan E. Lander and Charles J. Petrie, Concurrent Engineering: Research and Applications, Volume 4, Number 1, Pages 2-5, march 1996

185. **Agent-based engineering, the web, and intelligence**

Charles J. Petrie, IEEE Expert, Volume 11, Number 6, Pages 24-29, December 1996

186. **Agent-based approaches for advanced CAD/CAM systems**

W. Shen, F. Maturana, D. Norrie and J.P. Barthes, Proceedings of the fifth International. Conference On CAD/Graphics, Shenzhen, China, Pages 609-615, December 1997

187. **Mobile agents: Are they a good idea?**

D. Chess, C. Harrison, and A. Kershenbaum, In "Mobile Object Systems: Towards the Programmable Internet", Volume 1222 of Lecture Notes in Computer Science. Springer-Verlag, 1997

188. **The SkyBlue constraint solver**

Michael Sannella, Technical Report 92-07-02, Department of computer science and engineering, University of Washington, USA, 1993

189. **Geometric Modelling using octree encoding**

Donald Meagher, Computer Graphic and Image Processing, Volume 19, Pages 129-147, Academic Press Inc., 1982

190. **Extending the octree model to include knowledge for manufacturing**

G. Golbdogen and D. Ferrucci, Application of AI In Engineering Problems, 1st Int. Conf., Soat University, UK, April 1998

191. **OBBTree: A hierarchical structure for rapid interference detection**

S. Gottschalk, M.C. Lin and D. Manocha, Computer Graphics Proceedings, Annual Conference Series, Pages 171-180, ACM Siggraph, New Orleans, 1996

192. **Distributed software agents and applications**

J.W. Perram and J.P. Muller, 6th European workshop on modelling autonomous agents in a multi-agent world, MAAMAW'94, Odense, Denmark, August 1994
also Lecture Notes in Artificial Intelligence LNAI 1069, Springer-Verlag, ISBN 3-540-61157-6, 1994

Appendix A

Publications concerning the work presented

1. **Features as autonomous agents: an alternative paradigm for concurrent engineering**
   D. Jacquel, J. C. Salmon and F. Mill, ICMA 97. International Conference on Manufacturing Automation, Honk Kong, April 1997

2. **Solving the 3D localization problem for feature agents**
   Dominique Jacquel and Jonathan Charles Salmon, IFAC – INCOM'98, Volume 3, Pages 141-146, Nancy & Metz – France, June 1998

3. **Design for manufacturability: a feature-based agent-driven approach**
   Dominique Jacquel and Jonathan Salmon, accepted for publication in the international journal of the ImechE part B: Journal of Engineering Manufacture.

Paper published in ICMA'97 – Honk Kong – 28/30 April 1997

FEATURES AS AUTONOMOUS AGENTS: AN ALTERNATIVE PARADIGM FOR CONCURRENT ENGINEERING.

D. Jacquel, J.C. Salmon and F.G. Mill
Department of Mechanical Engineering
The University of Edinburgh, EH9 3JL, SCOTLAND

ABSTRACT

This paper describes a novel approach to the design of concurrent engineering systems by reversing the traditional view of such a system as a number of distinct but integrated modules operating on a data structure that is the product model. In this traditional view the data structure is a passive entity, and must be acted upon by modules such as design, process planning and NC generation.

In this new approach we imbue the model with intelligence (or at least a degree of autonomy), i.e., have an active model surrounded by passive expert modules which are capable of answering questions appropriate to their area of expertise. Indeed the model is allowed to be composed of any number of active agents, each responsible for an appropriate portion or feature of the model. Therefor ea multi-agent system is created in which each design feature is an agent that tries to successfully design, process plan and generate NC code for itself.

KEYWORDS

Feature Oriented Engineering, Agents, Multi-agent System, Active Model, Concurrent Engineering

1. INTRODUCTION

This paper focuses on a new approach for the design of a concurrent engineering system. This new approach takes its roots in the Artificial Intelligence field, and consists of applying the emerging multi-agent paradigm to the design of a feature based design system.

In the traditional approach, a concurrent feature based manufacturing system is an integrated set of software modules, such as modeller, geometric reasoner, process planner and NC code generator. They are all acting upon a passive data structure that is the product model, continuously enriching it so a design can evolve from the conceptual stage to a finished product complete with methods of production. Using this classical approach, the designer creates a product through a Design/ Evaluation/ Redesign loop, adding new features to the model before using one of several modules to evaluate the quality of the design against chosen criteria. This evaluation process can pinpoint flaws in the design, which can then be corrected before enriching the model with new features.

No matter how complex and powerful the modules used on the model are, the classical design process is not an interactive, concurrent activity but merely a client/server exchange between system and designer. The system will give answers to specific requests but is incapable of taking initiatives in the design process.

It is proposed to reverse this traditional approach, and create a system with an active product model and passive modules. This is achieved by applying the increasingly popular multi-agent paradigm to the system architecture. Indeed each instance of a design feature is an autonomous agent in the system. The product model therefore becomes an active community of feature agents that communicate and cooperate with one another in order to accomplish their goal: *engineer themselves.* Thus, the design module looks like a traditional design system from the user's point of view, but unbeknownst to the user, the roles have been reversed. The user firmly believes that they are in control and are designing some component for their own purpose. The product agent on the other

hand is 'using' the designer as an expert tool to fill in details about the design. In this way, user and product work in a symbiotic relationship to achieve a common aim.

An important factor is the interrelationship between the feature agents making up a complete product design. Certain features may intersect with each other or interact with each other in interesting ways that place considerable constraints on the manufacturing solutions for individual features. This problem is solved using two mechanisms. Firstly, all agents must be able to cooperate within the product model that they inhabit. The analogy here is one of a community and though each agent wishes for success as an individual, success of the entire community is paramount. Co-evolution techniques leading to emergent near-optimal solutions is an active research topic in Artificial Intelligence the results of which will be drawn on for applications in the concurrent engineering domain [1].

2. AGENTS

Formalising an agent definition is a difficult task, despite, or indeed because of, the vast amount of activity currently in the field. It is important distinguish the nature of agents at both the agent level and the community level.

2.1 Description of an Agent

Many recent publications related to agents and multi-agent systems (MAS) propose to give different definitions for agents [2][3][4]. At the entity level the most important features of a software agent are its co-operation and sociability aspects. An agent, can be thought of as an entity working inside a community and co-operating with other agents to achieve a goal. Rather than defining the nature of an agent, agency is indirectly described through properties required of agents.

- **Autonomy**: An agent can act with a certain range of autonomy and is capable of spontaneous actions. An agent has the capacity to plan its own actions and follow a self-prescribed schedule. Such autonomy can only be achieved through both synchronous and asynchronous actions.

- **Communication**: An agent must be able to carry bi-directional conversation with other agents in a language rich enough to allow it to express intentions and abilities to the community. Moreover, in order to comply with the autonomy property the communication protocol used by an agent should break the client/server protocol and permit peer-to-peer dialogues. [5]

- **Co-operation**: Using their communication abilities, agents should be able to initiate dialogues with one another in order to achieve their goals. This co-operative behaviour should lead to improved reactivity of the system and additionally, better interaction with users.

2.2 A Multi-Agent System (MAS)

Only considering the notion of agency at the agent level ignores the social dimension of agents. A social software agent is only useful if living inside a community of agents. In such a community, the asynchronous nature of each agent leads to exchanges of messages and decision/actions throughout the system. This asynchronous, peer-to-peer activity between individuals inside the system results in an overall multi-goal, co-operative search for an optimum solution.

Unlike a traditional procedural synchronous system, each feature agent initiates a dialogue only on an asynchronous basis, when it needs to solve a problem or improve its fitness against its goals. This allows the system to only use its processing power on critical parts of the design model, resulting in improved resource allocation.

2.3 Agents and Concurrent Engineering

Concurrent engineering (CE) proposes to make product creation a faster and more efficient process by allowing traditional sequential tasks such as geometry design and process planning to take place simultaneously. By parallelising the different tasks of the engineering process, CE brings early feed back to each task, avoiding costly and time-consuming redesign phases. The multi-agent paradigm and techniques have already been applied in a number of ways to CE [6].

Agents can be used as a mean of integration between the existing engineering tools by using autonomous interface agents between design and process and manufacturing applications [7]. In this approach agents provide their communication and information sharing capabilities to create a dialogue between design and manufacturing, bringing early feedback about the manufacturability of a product being designed.

Using more than interface agents [8] proposes integration of design, manufacturability analysis, incremental process planning, dynamic routing, and scheduling, using both feature agents and module agents (geometric interface, design agent, part agent and machine agents).

This work has arisen out of a previous project entitled *Simultaneous Engineering System for Applications in Mechanical Engineering* (SESAME), (BRITE/EURAM 0565). The goal in SESAME was a Simultaneous Engineering Workstation (SEW) and attempted to unify the tasks of Computer Aided Design, Process Planning and NC Generation in such a way that they could be used by a single engineer on a single seat in a standard environment. This goal SESAME achieved with a significant degree of success [9], but the tasks were still performed in a sequential manner as is evident from the architecture in Figure 1. The system was not truly concurrent, but achieved greater integration than earlier process planning systems.

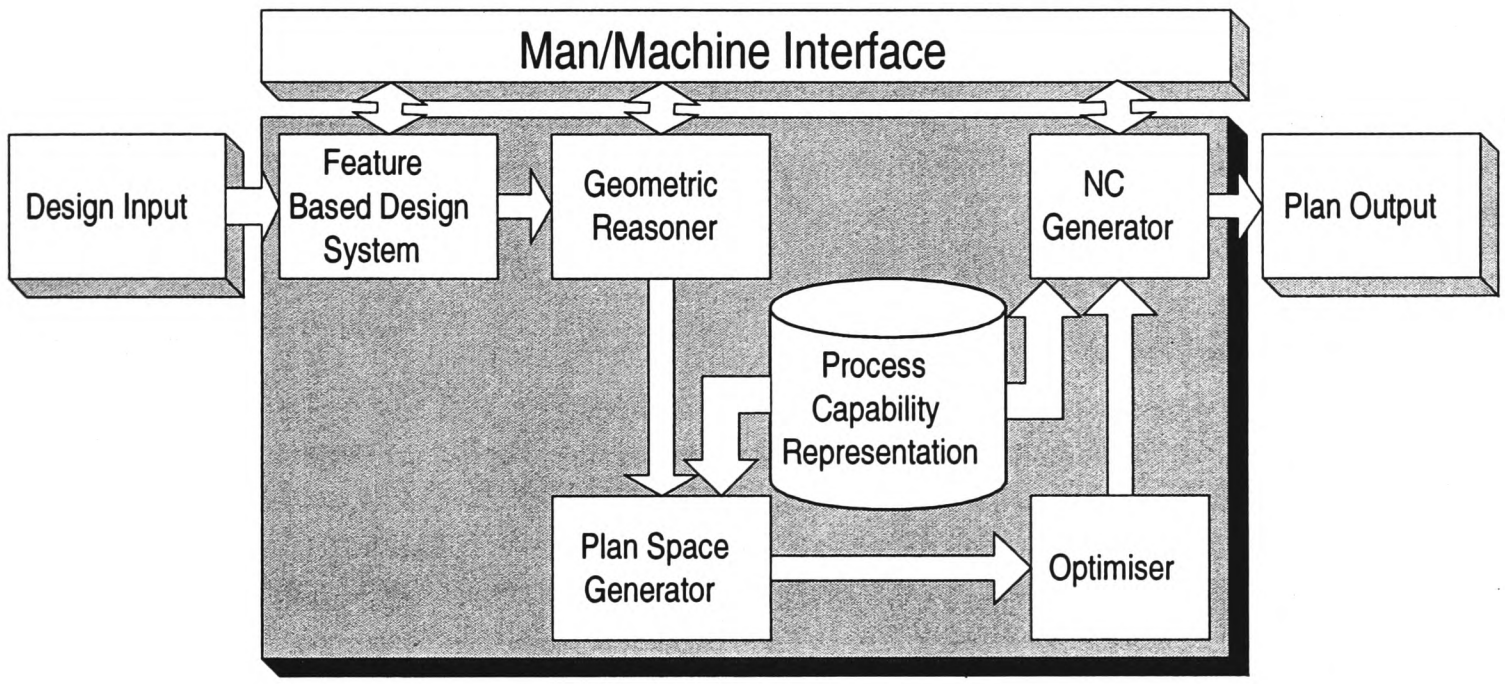


Figure 1: SESAME Architecture

A subsequent proposal in order to increase the concurrency of the system was to build intelligent agents capable of incremental design, process planning and NC generation. This coarse grained approach (Figure 2) can be seen in a number of systems [10], and is a valuable approach when trying to integrate existing products into a concurrent agent based systems.

The Edinburgh 'Features as Agents' approach (Figure 3) uses a fine-grained model with lightweight agents acting for the features in the design and being aided by expert assistant modules such as incremental design, process planning and NC generation systems. This method is not suited to adapting existing commercial systems, but the knowledge gained in the group from SESAME and other projects is present to enable existing modules to be rewritten and adapted.

This fine-grained approach allows solutions to the current design problem to be developed all the time in the dead time between the human's design decisions. The fact that many of these solutions may be unsatisfactory in the finished design is unimportant as the system always holds as complete a picture as it can at any point, and after the last design decision is made, little work need be done to take the current plan and adapt it to the final design (in the majority of cases). In addition the designer is kept constantly aware of the downstream implications of their design.

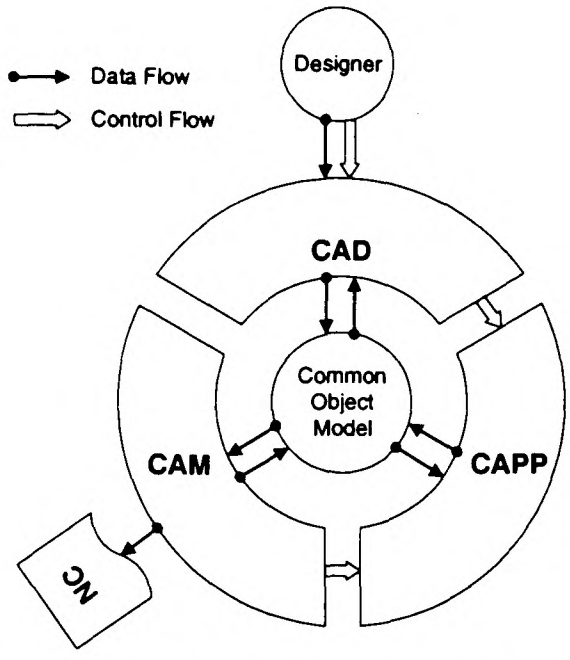


Figure 2: Coarse Grained Agent Model

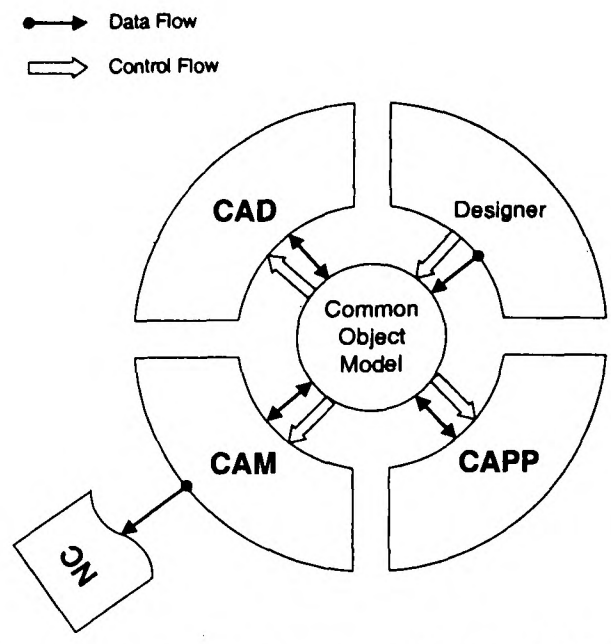


Figure 3: The 'Features as Agents' Model

3. THE EDINBURGH 'FEATURES AS AGENTS' PROPOSAL

The SESAME project was a major improvement over traditional systems, offering a highly integrated set of tools under a single interface allowing a user to cover the complete design process with rapid feed back about manufacturability and permitted the generation of NC code. However the design process, even though integrated, remained sequential rather than concurrent.

The Edinburgh proposal advances the benefits of SESAME by replacing the static product model with an active representation. This can be achieved with the multiagent paradigm applied at low 'fine-grained' level of design features. This radical switch underneath the interface is bringing new benefits to the system. It is also an efficient way of applying and solving design constraints as they can be made part of each agent's goal. It should also ensure quality models at all time and allow complex questions to be answered quickly as agents are constantly working on behalf of the user to find an optimum solution. This constant activity generates much redundant information, but this information is generated in otherwise idle CPU time, increasing the system's apparent performance.

3.1 Small example

To illustrate the proposal, a simple example of how the system works during the design of a small prismatic component is now presented.

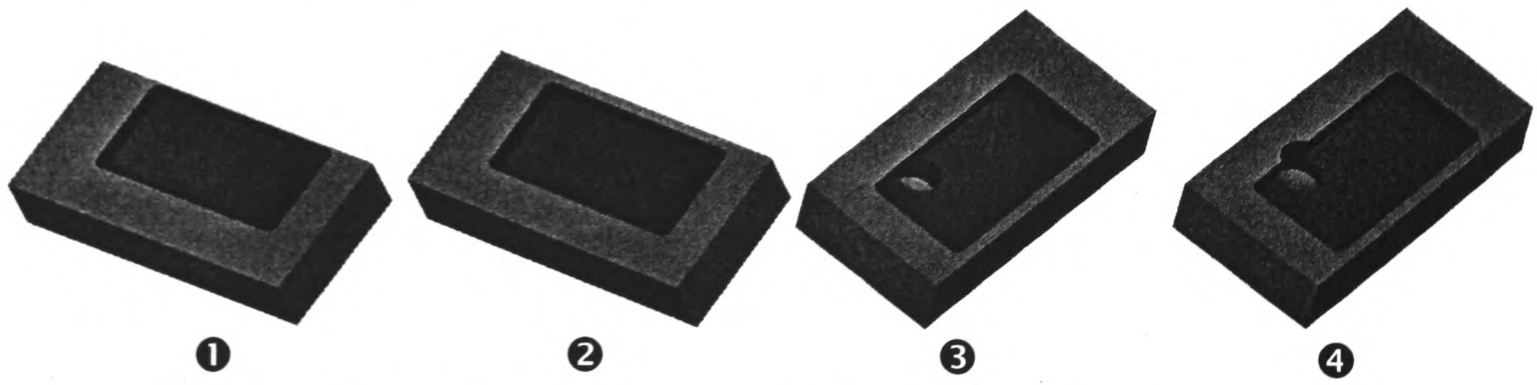


Figure 4: An example component design sequence

Table 1 below shows the evolution of the system during the design process pictured in Figure 4 at three different levels. The user level reflects the designer interaction with the system. The model level represents the product state in terms of geometry. Finally the agent level focuses on the state of and communication between the agents populating the model.

This small example illustrates two important concepts not found in traditional CAD systems. It displays automatic resolution of a thin wall problem within the constraints defined by the user (*). It also illustrates the increased interactivity by allowing an agent to initiate dialogue with the user in the case of an unresolved problem (**).

Table 1 Agent Communication during Design Example

User level	Model Level	Agent level (① shows inter-agents communication)
Creation of blank block with tight size and position constraints	Untouched blank	**block**: I'm the only agent in the model … sleep
Add new pocket with normal position constraints	❶	**pocket**: I'm not alone in the model. **Pocket**①*Broadcast*: New pocket at position xyz **block**: thin wall problem, but I can't move/resize **block**①*pocket*: you're creating a thin wall
	❷	**pocket**: I'm moving to new position xyz (*) **pocket**①*Broadcast*: pocket at new position xyz **pocket**: no more problem … sleep **block**: no more problem … sleep
Add new hole with tight constraints on position	❸	**hole**: I'm not alone in the model. **Hole**①*Broadcast*: New hole at position xyz **block**: I can't guarantee access, I can't move/resize **block**①*hole*: access problem block position xyz **pocket**: the new hole interacts with me. **pocket**①*hole*: there is a pocket at position xyz **hole**: try to solve access problem, but I can't move/resize **hole**①*pocket*: Can you move to new position xyz? **pocket**①*hole*: I can't comply with request. **hole**: I can't solve this problem. **hole**①*user*, I have an access problem. (**)
Receive message from hole. Modify hole position	❹	**hole**①*Broadcast*: hole new position xyz **block**: no more problem … sleep **pocket**: no more problem … sleep

3.2 Current state of work

Using the Feature Based Design knowledge gained in SESAME, a new design tool is being implemented as a multi-agent system. It takes a fairly simple subset of the problems to be solved during the conception process in order to test the validity of our approach.

To an external user, the new implementation appears little different from the previous Feature Based CAD system. It allows the user to add negative (material removal) features to a blank to obtain the desired design. Underneath the user interface, however, radical changes have taken place. Where there use to be a passive product model on which software modules were acting, there is now a community of agents tirelessly working together to reach an optimum solution.

Each time the user adds a new feature to his design, they actually gives birth to a new software agent that join the existing community inside the system. As soon as this new feature/agent enters the MAS, it can start following its own plans and interact with other agents in order to satisfy its goals. It is this underlying community of agents that represent the living product model.

The current MAS is limited to geometric reasoning such as thin walls detection but it already shows its potential by immediately detecting problems during design. For example, if the user places a new hole too close from another one, creating a thin wall, the system will immediately detect it and take action to solve the problem. This could lead to one of the agents deciding simply to move slightly aside or reduce its diameter (subject to previously supplied constraints), or ask the user for an alternate solution. The system also delivers better performance in terms of response time when adding new features to an existing design because it is incremental by nature. It doesn't need to re-analyse the entire design against the new feature, the new agent interacts locally with existing agents only. Of course a snowball effect is always possible when adding a new feature to the design but this only happens if agents are insufficiently constrained, or if the new feature interacts with many others.

3.3 Future work

The first addition to the current implementation will be to add non-feature agents for conflict resolution and solution proposal. These new agents should share the same structure as the feature

agent but provide more complex computation power to propose solutions to problems our light-weight feature agents are unable to solve. A second addition will be expert modules to handle support for process-planning and NC code generation to our current agents. This should not be a major difficulty thanks to the modularity of the Edinburgh approach. Adding support for these tasks also requires adding new knowledge/abilities to our existing agents.

It is already possible to express constraints local to a single agent, the next step is to work on constraints propagation inside the agent community forming the product model. It is also intended to investigate the feasibility of applying the multi-agent paradigm to a constraint solving feature based design system that would automatically generate design solutions. Lastly, we will investigate solutions to the problem of combinatorial explosion that we can see looming in the system.

4. CONCLUSION

This new approach, consisting of turning the system upside-down, making each feature an active element of the system has, so far, shown potential. The constant activity of the agent community radically changes the way the system behaves. The active model, always in search for an optimum solution, ensures a quality model at all stage of design. Leading to several major improvement from the traditional design system architecture.

One could argue that constantly analysing the model generates tremendous load on the computing resources but the MAS provides a better use of computing resources compare to the traditional approach. Indeed, it is believed that, despite the large increase in term of number of operations, the active model provide better overall performances by diluting the calculations along the entire design process.

The Design-Evaluation-Redesign loop no longer needs to be performed by the user. This process is achieved individually by each agent in the system during the design phase. Being able to detect potential problems at early stages through this constant self-assessment of the model, the multi-agent approach reduces costly redesign. The active model is also an efficient way of applying constraints to the design since the constraints can be embedded inside each agent. Finally a higher interactivity during the design process is achieved by enabling any agent to initiate a dialogue with the user.

5. REFERENCES

[1] Husbands P., Mill F., 'Simulated Co-evolution as the Mechanism for emergent planning and scheduling', Proceedings of the Fourth International Conference on Genetic Algorithms, Pages 264-270, Morgan Kaufmann Publishers, San Diego CA, July 13-16, 1991.

[2] Wooldridge M, Jennings NR, 'Intelligent Agents: Theory and Practice' , Knowledge Engineering Review, 10(2):115-152, 1995.

[3] Genesereth MR, Ketchpel SP, 'Software Agents', Communications of the ACM 37(7): 48-53, July 1994.

[4] Franklin S, Graesser A, 'Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents', Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages, Springer-Verlag, 1996.

[5] Petrie CJ, 'Agent-based Engineering, the Web, and Intelligence', IEEE Expert. 1996.

[6] Brown DC, Lander SE, Petrie CJ, 'The Application of Multiagent Systems to Concurrent Engineering', Concurrent Engineering: Research and Applications, 4(1): 2-5, March 1996, Technomic Publishing Co Inc, 1063-293X

[7] Frost RH, Cutkosky MR, 'Design for manufacturability via agent interaction', ASME Design for manufacturing Conference, paper number No 96-DETC/DFM-1302, Irvine CA, August 18-22, 1996.

[8] Balasubramanian S and Norrie DH, 'A Multiagent Architecture for Concurrent Design, Process Planning, Routing, and Scheduling', Concurrent Engineering: Research and Applications, 4(1): 7-16, March 1996, Technomic Publishing Co, 1063-293X

[9] Mill F. G., Naish J. C., Salmon J.C., 'Design for machining with a simultaneous-engineering workstation', Computer-Aided Design, 26(7): 521-527, July 1994.

[10] Wunderli M, Norrie M; Schaad W, 'Multidatabase agents for CIM systems', Int. J. Computer Integrated Manufacturing, 9(4): 293-298, 1996, Taylor & Francis, 0951-192X

Published in INCOM'98 - June 24-26, 1998 - Nancy & Metz (France)

SOLVING THE 3D LOCALIZATION PROBLEM FOR FEATURE AGENTS

D. Jacquel and J.C. Salmon

Department of Mechanical Engineering, The University of Edinburgh
Sanderson Building, King's buildings, Edinburgh EH9 3JL, SCOTLAND

Abstract: A feature-agent based Design and Manufacture System is described. A particular problem associated with agent software, that of assigning spatial locality to agents and allowing inter-agent communication based on that locality is discussed. A solution utilizing Beliefs-Desires-Intentions (BDI) agents is described. *Copyright © 1998 IFAC*

Keywords: Agents, Concurrent Engineering, Design, CAD/CAM

1. INTRODUCTION

Feature-based modeling brings increased functionality to CAD/CAM and helps generate models that are more complete, robust and ultimately more manufacturable. For a feature-based model this manufacturability is highly dependent on the way the features within the model interact geometrically. Indeed, accessibility, proximity and collisions are crucial notions in manufacture that are linked to feature interactions. To achieve greater concurrency in the engineering process it is important to solve these 3D positioning problems early in the product life cycle. The use of autonomous software agents to embody design features in order to address inter-feature interaction problems during the design stage is proposed.

1.1 Geometric reasoning in CE

The challenge for a fast transition from product concept to manufacture is largely being won or lost during the early design stages. The adoption of a feature-based design system is a first step toward ensuring machinable models as features provide a natural means to associate domain knowledge like manufacture processes with object representation (Vandenbrande and Requicha, 1993). It is recognized that designers require efficient design validation tools in order to meet the manufacturability criteria that are expected from them. Such design validation tools perform geometric reasoning on the product model to detect problems and inconsistencies. Needless to say that the performance of such a tool can greatly influence the smooth operation of all downstream phases of the product life cycle.

1.2 Multi-agent systems

Despite being the topic of numerous publications, the concept of agency still doesn't have a definition that is unanimously agreed on. A number of required properties however seem to be widely recognized in the scientific community. For the software agents that interest us, the following features are accepted to define an agent (Wooldridge and Jennings, 1995; Genesereth and Ketchpel, 1994): *Autonomy*, *Communication* and *Co-operation*.

1.3 Multi-Agent Design Systems (MADS)

Most of the work investigating the application of multi-agent technology to engineering design uses coarse-grained agents as autonomous communicative entities added to existing design systems. These

agents provide both their communication and their knowledge sharing abilities to create data exchanges between the different design stages on behalf of their users. As part of the GNOSIS research program however, (Balasubramanian and Norrie, 1996) propose a fine-grained approach to the agent-isation of a feature based design system to integrate manufacturing design and shop floor control. They propose to associate each design feature with an autonomous agent. These feature agents initiate dialogues with other agents (machine tools, stock managers, etc.) in order to achieve greater concurrency in the product creation process.

2. THE "FEATURE AS AGENT" PROPOSAL

2.1 A fine grained "agent-isation"

Like (Balasubramanian and Norrie, 1996) our 'feature as agent' proposal also offers a fine-grained agent-isation of the manufacturing design system. It turns each feature into an autonomous software agent, transforming the usual passive data oriented product model (CAD files) into an active decision oriented community of agents. Our active model is able to exercise a degree of autonomy in modification of the design itself.

The granularity of our approach, which focuses on a feature level, is justified by work being carried out in the feature based manufacturing design domain (Mill, et al., 1993; Shah and Mantyla, 1995). Indeed these results show that most problems encountered during design, process planning or manufacturing, are a consequence of the way features relate to each other (Vandenbrande and Requicha, 1993). The resolution of conflicts arising at an early stage permits creation of manufacturable models. Our approach proposes a very fine-grained agent-isation to support inter-feature conflict resolution during the design stage.

2.2 Expected advantages

In addition to the usual and expected benefits of agent systems such as scalability, extensibility, fault tolerance, and modularity, the fine grained approach applied at the design-feature level brings several new qualities to the system.

Increased interactivity and responsiveness
The agent-ified design system should also offer greater interactivity. Indeed an agent is able to initiate dialogue with any other agent in the system when necessary. On a peer to peer communication scheme, autonomous agents in the model are able, for instance, to point out a potential problem to the user. This information is given to the user without a request being necessary.

Increased system autonomy
Each feature inside the model is now an active autonomous entity, it can automatically perform certain tasks on behalf of the user. In particular, each feature can now try to resolve certain types of existing conflicts, leaving the user free to focus on the design process.

Real time analysis of the design
The product model that used to be a passive data structure acted upon by sequential modules is now transformed into an active community of agents that initiate dialogue with one another. The most obvious consequence to this radical change is a model that endlessly assesses and, if necessary, corrects itself on behalf of the designer. Each agent living inside the model runs autonomously and follows its own plans to reach its goals. These individual goals are aimed both at an individual level: "*I must manufacture myself*", and on a global level: "*global product manufacturability is paramount, I should degrade gracefully*". Following these simple principles, each feature/agent inside the model, constantly probes its surrounding and tests itself against collected data to detect any conflicts. Therefor the MADS provides true real time design analysis through its modular parallel architecture.

Problem focused system
Not only does each agent achieve real time analysis of itself, it also does it locally rather than globally. With our agent based approach the system is naturally incremental as all analysis and computations are performed locally at the feature/agent level. Thanks to its new architecture, the system only focuses on critical areas of the design. It naturally avoids wasting resources testing unmodified parts of the model because agents only respond to modifications in their local environment. Most of the available activity in the product model is therefor directed at detecting/solving existing problems rather than re-assessing the good portions of the design.

2.3 Potential Drawbacks

Exponential communication load
As for most multi-agent systems, the communication combinatorial explosion lurking in our new architecture is probably its Achilles heel. However, it can be minimized, by taking advantage of the systems new properties. In particular, the ability of a feature agent to focus only on unwanted interactions with its immediate surroundings should provide a way to create dynamic communication clusters inside the model. Moreover, each agent updates and manages its own belief database, which greatly decrease redundancies in information exchanges.

System Responsiveness vs. Model Stability
The fine-grained agent-isation gives autonomy and freedom to design features, allowing the model to work out solutions autonomously, on behalf of the user. Unfortunately, too much freedom and initiative from features could also lead to unstable design solutions. Typically, the symptom of this instability (or hypersensitivity) is the 'explosion' of a perfectly viable model by the adjunction of one feature agent creating strong interactions within the existing community. This newly introduced agent causes reactions/adjustments in the community that can cause the loss of the existing partial solution. A balance between the desired system responsiveness and the necessary stability of the produced model has to be found. A realistic way to achieve it is to constrain all features to restrict their freedom of movement. In order to prevent the described model hypersensitivity, these constraints can be tightened on part of the design to preserve partial solutions

Saving and loading models
With our approach the product model being designed is no longer a passive data structure holding the geometry. It has become much more than that. It's a community of agents working together to reach both an individual and a common goal. Each agent in the model builds its own representation of its environment. Indeed, it collects and records information useful to its activity such as addresses of agent it previously interacted with or geometry of its surrounding. Because of this additional information collected by each agent in the model, problems concerning the product model serialization (storing and retrieving) arise. What should the archived version of the model contain?

In a conventional design system, the answer to this question is straightforward and easy to implement. Because the model is a passive data structure being act upon by modules, it is easy and efficient to store it as a file. Indeed, all the semantic of the model is captured by the geometry of the model. With our agent approach however, there is no simple solution to this problem. On one hand, storing only the geometric data held by each agent represents a tremendous loss of knowledge to the living agent community. On the other hand, serializing the entire model would create obvious redundancy and would not be easy to implement.

3. Implementation: Simplified Multi-Agent Feature Based Design System

A number of preliminary implementations of our multi-agent design system have already been completed and tested. They were not meant to be complete or efficient. These test implementations were realized in order to compare the different possible type of agents eligible to embody our design features. These multi-agent architectures have been implemented in C++ under Windows NT.

3.1 KQML compliant

Coordination can be achieved without communication with real world agents such as robots. This structural coordination or organizational structuring as defined by (Nwana, et al., 1997) can be achieved but in the case of purely software agents like our features, communication is mandatory as it is used instead of sensors as a medium for information collection.

The Knowledge Query and Manipulation Language (KQML) is becoming the *de-facto* agent communication language standard. It satisfies most of the requirement for an ACL (Mayfield, et al., 1996) and was adopted as the common ACL in our test design systems. Only a small subset of the defined communication acts (called performative in KQML) had to be implemented to support our test agents. Unknown performatives can be handled by the default *sorry* act, which allow integration in the same system of agents with various level of KQML support.

3.2 Communicative agent architecture

In order to support high-level communication services in a KQML compliant form, our agents have to offer low level communication protocols.

Complex geometric models are expected to generate very high workload on our new multi-agent architecture. Therefor our implementation choices were dictated by the search for good performance in terms of responsiveness and speed. Our test system was implemented as a stand-alone Windows NT applications hosting all agents as threads of execution. This "centralized" approach has the advantages of being economical in term of resources needed and to allow our work to focus on the agent issue rather than on the object distribution problems.

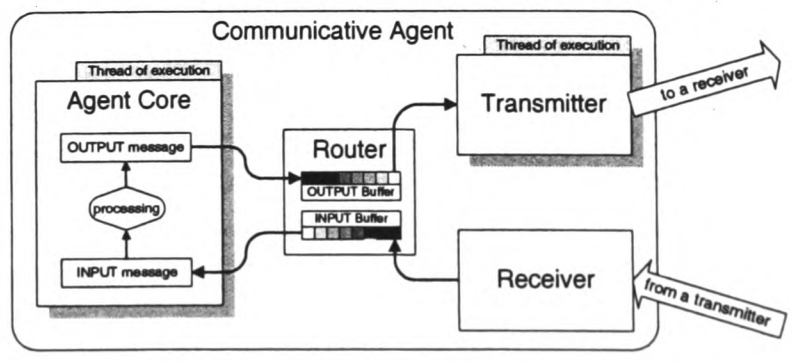


Figure 1: Agent's communication architecture

As shown on Figure 1, each communicative agent is actually composed of two threads of execution

running in parallel. This way the agent doesn't have to interrupt its activity to receive or send messages. This provides complete separation between the agent core and communications. The agent just connects to a router that provides standard functions to access its message buffers. This way, support for new communication protocols can be easily added by modifying receiver, transmitter and part of the router.

3.3 Reactive agents

Reactive agents are programmed to automatically respond to certain situations with a predefined action or set of actions. They do not have a mental state of any sort. Therefor, they do not have any knowledge about their surrounding nor do they remember previous events.

A first test implementation of our multi-agent system was written to assess suitability of such a simple agent to perform complex task involved in engineering design. The test agents were implemented and embodied co-planar 2D circles. They were programmed to avoid colliding with each other. This simplified problem is related to hole collision detection and solving in 2½D prismatic design. Each circle agent in the system reacted according to two simple rules.

- It broadcasts any change of its position and dimension.
- If a position message is received and a collision is detected, the agent changes its position to solve the detected collision.

The test run of this system demonstrated a very unstable behavior almost always leading to live-lock (Nwana, et al., 1997). These locks lead to a situation where a group of agents endlessly change their positions, chasing one another other across the plane. The introduction of a small degree of randomness in the direction taken by a colliding agent decreased the frequency of these live-locks but never totally eradicated the problem. Tests also showed that addition of one agent in a perfectly stable model could provoke its 'explosion'. Also, it was obvious that when solutions were reached they were sub-optimum.

These behavioral problems encountered by our reactive system are the symptoms of a deeper illness. Basic reactive agents have no mental state, and therefor have no knowledge of their environment, and no recollection of their past actions. This explains how live-locks are so easily reached. Because agents share the code of their behaviors, they will take identical decisions when confronted to identical situations. Since they have no mental state they can get locked into live-locks where they constantly run into each other while trying to avoid exactly that. Also, the lack of knowledge about their environment will makes agents take uninformed and selfish decisions that can create more conflicts than they solve, leading to the described model 'explosions'.

3.4 BDI agents (Intentional agents)

Our first test implementation made it clear that reactive agents are not good enough to achieve the complex behavior expected from a design feature. In order to avoid live-locks, our agent will have to possess a certain amount of memory in which to store records of previous events or a representation of the known external environment. Consequently, the second step in our testing was to add mental states to our agents in order to brings stability to the system.

The BDI (Beliefs Desires Intentions) architecture adds mental states to agents. It separates these mental states into distinct categories.

- **Beliefs** represent the dynamic knowledge of the agent. Some of these beliefs will be provided initially but most of them will collected and revised (Malheiro, et al., 1994) by the agent during the course of its live.
- **Desires** represent what the agent has to achieve; they encode the desired behavior of the agent.
- **Plans** are the know-how of the BDI agent. They encode the way the agent will act in response to given conditions.
- The agents generate **intentions** according to the plan that is executed. These intentions lead to real actions by the agent.

At all times, the BDI agent can receive data from it environment to keep the Beliefs up-to-date. Autonomously, the behavior engine endlessly compares its beliefs about the present situation and its desires. If a mismatch is detected, the Plan database is searched for a suitable set of actions. The chosen plan generates a set of intentions that will be turned into actions unless they are discarded while still at the intention stage. The advantage of this architecture is that the agent can acquire the information about how best to achieve its goals during the plan execution. A BDI agent is also focused on offering adequate response to situations when other planning agents often commit themselves too strongly to their plan. The BDI approach therefor permits agents to always be in phase with their environment by strongly linking their action with the present situation. Inspired by the feature validation rules described in (Vandenbrande and Requicha, 1993) our BDI feature agents are motivated with the following desires.

- **Presence**:

Principle: Each feature in the model must contribute to the design by producing at least one surface on the

finished part boundary.
Implementation: Each feature keeps track of all known intersections. A presence counter is kept up-to-date at any time that counts the number of intersections with features of opposite materiality. A desire is set that the presence counter ≥ 1.

- **Collision**:

Principle: Certain inter-feature intersections must be avoided for a viable design to be reached. The physically impossible intersection between two positive features is one of these.
Implementation: A collision counter is added to the feature agent. It counts the number of undesired intersections with features of same materiality. A desire is set that the collision counter = 0.

- **Proximity**:

Principle: If two negative features are too close to each other, they might create a thin wall problem that render the machining impossible.
Implementation: A proximity counter is added to feature agents and kept up-to-date as art of its beliefs. A desire is set that the proximity counter = 0.

- **Accessibility**:

Principle: For a negative feature to be machined, the cutter has to be able to access it. Accessibility to a feature can be obtained either directly when the feature emerges to open space or can be obtained indirectly through another accessible feature. Note that different orientations and entry points can also create accessibility to a feature.
Implementation: A counter is added to each feature agent that keeps track of all possible machining accesses to itself. A desire is set that the access counter ≥ 1.

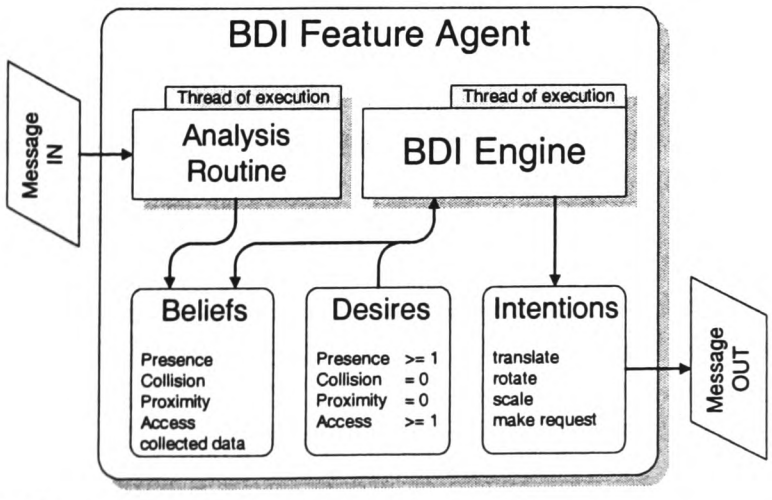


Figure 2: BDI feature agent internal operations

Figure 2 shows the general operation of our DBI feature agent. When the ontology of the incoming message is ***geometry*** it is directed to the main analysis routine. That routine runs simultaneously with the BDI inference engine, and contains all the geometric problem detection knowledge of a feature agent. It tests the geometric data received against the feature's geometry. If interactions are detected, the beliefs are incrementally updated to reflect the new knowledge of the situation. The beliefs are dynamically updated to reflect the environment of the feature. This way the BDI inference engine running autonomously in its own thread of execution always works with consistent beliefs. It infers intentions by checking for mismatches between the agent's four desires and their corresponding beliefs. When mismatches are detected, a suitable plan is picked and applied to generate intentions. The BDI agent turns these intentions into actions (move, scale or rotate) and sends messages to other agents about geometry changes or to make requests.

By adding a mental state to our agents, the BDI approach brings the desired stability that is lacking with the purely reactive agents but still offers good performances in term of system responsiveness. A BDI agent possesses a belief database that enable it to build and update its own representation of the world. This memorization ability impacts greatly on the quality of the decisions made. Indeed, having a knowledge of its present situation, a feature-agent can assess the potential consequences of its decisions before acting. In case of problems involving a large number of features, the model doesn't "explode" and reaches a solution fairly quickly. However, the BDI approach is not the universal panacea and live-lock situations still occasionally arise in the product model. They can usually quickly be solved by manually forcing a solution or by inserting a new feature that acts like a key (it induces new reactions in the locked agents that breaks the live-lock). In fact, (Nwana, et al., 1997) describes the live-lock situations as a symptom of the lack of cooperation between agents.

4. FURTHER RESEARCH

At the light of our preliminary results, a number of paths have been identified that will need further investigation.

Negotiation capable agents
"Successful co-ordination is a key design objective for most multi-agent system builder (Nwana, et al., 1997)". Without negotiation between agents, the designer has to rely on structural organization to achieve the community behavior he wishes to create. Such systems are prone to instability, get caught in live-locks and don't converge to an optimal solution.

Support for complex geometry
In order to deal with real world complex geometry, our feature agents need a compact representation of their environment and efficient 3D operators to work on it. Use of octree representation (Meager, 1982) to locally store geometric knowledge collected by each feature agents is planned. Octrees offer a very

efficient solution to interference detection problems and facilitate CSG operations. The hierarchical representation allows algorithms to operate at a level appropriate to the task.

Non-feature service agents
Providing they support the common ACL, non-feature agents, acting as service providers to the feature agent community in the model can be added with ease. Such agents provide complex conflict resolution or design fitness assessment or make original solution proposals to problems resisting the community efforts.

Adaptive complexity of 3D reasoning
Due to the performance constraint on our lightweight agents, the generic feature agent was given an efficient internal routines to manipulate the geometry of its Bounding Box but not of its detailed geometry. The idea is that most of the interaction problems can be solved or partly solved at the bounding box level, and so is economic in its use of resources. The agent relies on dedicated service agents to compute complete agents geometry when the bounding box abstraction is insufficient to reach a solution.

Collaborative agents
The next stage in our work will be to investigate potential collaboration techniques that would suit our architecture constraints. The collaboration algorithm used by feature agents will have to be economical in term of communication load and to be efficient both in size and speed. In a similar way to complex geometry support, dedicated service agents will be used to balance the lack of complex collaboration inside the feature agents.

Communication load, possible solutions
To prevent degradation of the system performance when models get more complex, ways to reduce the likelihood of a combinatory explosion of messages must be found. A promising method relies on creating dynamic agent clusters inside large models to dramatically reduce the overall system load. The inter-feature communications are strongly related to geometric intersection and proximity so the clusters can be identified on geometric basis.

CONCLUSION

A new approach has been presented that allows the designer to focus on component design because feature agents living inside his model can autonomously carry out some 3D localization operations on his behalf during the design process. Our architecture based on the KQML agent language and the BDI agent model has been implemented that support that autonomous 3D feature positioning.

The fine-grained feature agent-isation brings much more to engineering design systems than the gross-grained approach. It offers increased interactivity, responsiveness, autonomy, scalability and upgradability. More importantly, the very nature of the product model is radically changed. The model is no longer a passive data structure on which operations are performed by the designer but an active community of agents that can take initiatives concerning itself.

REFERENCES

Balasubramanian S. and Norrie D. H. (1996). A multiagent architecture for Concurrent Design, Process Planning, Routing and Scheduling. In *Concurrent Engineering: Research and Applications.* **Vol. 4, No 1**, P. 7-16

Fisher Klaus, Muller Jorg P.and Pishel Markus (1995). A pragmatic BDI Architecture. In *Intelligent Agents II, LNAI 1037* Springer-Verlag

Genesereth Michael R. and Ketchpel Steven P. (1994). software agents. In *Communication of the ACM.* **Vol. 37, No 7**, P. 48-53

Malheiro Benedita, Jennings N. R., Oliviera Eugenio (1994). Belief Revision in Multi-agent Systems. In *ECAI 94. 11th European Conference on Artificial Intelligence.*

Mayfield James, Labrou Yannis and Finin Tim (1996). Evaluation of KQML as an Agent Communication Language. In *Intelligent Agents II LNAI 1037* Springer-Verlag

Meagher Donald (1982). Geometric Modelling using octree encoding. In *Computer Graphic and Image Processing.*, **Vol. 19**, P. 129-14

Mill F. G., Salmon J. C. and Pedley A. G. (1993). Representation problems in feature-based approaches to design and process planning. In *International Journal Computer Integrated Manufacturing.* **Vol. 6, No 1**, P. 27-33

Nwana H. S., Lee L., Jennings N. R. (1997). Co-ordination in multi-agent systems. In *Software agents and soft computing LNAI 1198* Springer-Verlag

Shah Jami J. and Mantyla Martti (1995). *Parametric and feature-based CAD/CAM.* John Wiley & Sons, inc.

Vandenbrande Jan H. and Requicha Aristide A. G. (1993). Spatial Reasoning for the Automatic Recognition of machinable features in solid models. In *IEEE Transaction on pattern analysis and machine intelligence.* **Vol. 15, No 12**, P 1269-1285

Wooldridge Michael, Jennings Nicholas R. (1995). Intelligent agent: Theory and Practice. In *Knowledge Engineering Review.* **Vol. 12, No 2**, P. 115-152

Accepted for publication in the ImechE journal of Engineering Part B: Engineering manufacture

Design for Manufacturability: a Feature-Based Agent-Driven Approach.

Dominique Jacquel and **Jonathan Salmon** (corresponding author),
(D.Jacquel@ed.ac.uk & J.C.Salmon@ed.ac.uk),
School of Mechanical Engineering, The University of Edinburgh
Sanderson Building, King's Buildings, Edinburgh EH9 3JL

Abstract: Extensions to mechanical feature-based design and design for manufacturability are presented that adopt the increasingly recognised multiagent paradigm. This approach uses autonomous agents to implement each form feature inside the model being created, thus leading to the creation of a new type of active product model. Designers add new features to their designs by populating a living community of agents that construct the model, therefore creating an active product program as opposed to the traditional passive product data.

Feature agents are self-scheduled autonomous entities, able to exchange data with one another and with a degree of self-control over their own geometric data. Using simple manufacturability criteria, each feature agent endlessly applies its embedded knowledge to ensure its own local manufacturability. A global behaviour emerges from the agents' activity inside the model that optimises the global manufacturability of the component being designed.

Keywords: CAD/CAM, feature-based design, agents, manufacturability analysis

1 Introduction

In a typical CAD environment, the design progresses through a design-analysis-redesign loop. The analysis performed in this loop might be a functional analysis, or a stress analysis for example. Although the ever-increasing computing power available to the designer allows the inclusion evermore knowledge inside that loop, it remains a human driven loop. The computer performs complex tests on the design and provides the designer with high level results. The designer uses these results and experience to modify the design before resubmitting it to the computer. A natural progression is to replace, where possible, the weak human element of this loop, where human weakness is defined only in terms of speed and availability, rather than in terms of adaptability or quality of design.

The design analysis of interest here is manufacturability analysis. In the context of increasing global competition, it is of crucial importance to optimise time-to-market for manufactured products. Timely manufacturability analysis is an important tool to further this goal; a domain where computers can equal or better humans as manufacturability is strongly linked to the unambiguous geometric properties of the design and is so suited to automation.

The manufacturability of a design is the probability that it can be produced given a set of available machines, tools and processes. The manufacturability optimisation considered here does not include the generation of detailed process plans, nor cost estimation. In fact, the presented work is located in a phase immediately prior to process planning. This approach is a novel way of optimising the quality of the design before it is sent to the process planning system, avoiding the waste of resources involved in process planning unmanufacturable designs.

2 Overview of the approach

The multiagent paradigm is emerging in the artificial intelligence domain [**1, 2**]. It is an extension of the Object Oriented framework and the self-organisation of simple entities into complex systems. This self-organisation is achieved by emphasising the role of peer-to-peer communication between agents and allowing for autonomous decision making within each agent. The parallel with social insect colonies exemplifies how simple entities with no global view of a problem can self organise into systems capable of performing complex tasks. This organisation does not rely on centralised control, but rather on the emergence of a global coherent behaviour from the interaction of multiple simple and similar entities.

2.1 Delegation and trust

Agent based computing is all about task delegation. As agents able to take autonomous decisions, they are a powerful mechanism to deal with certain classes of problems. Their ability to schedule their own agenda within an application makes them ideal candidates to assist the human user in simple or repetitive tasks. Before delegating a task to an automated system, one has "to balance the risk that the agent will do something wrong with the trust that it will do it right" [**3**]. Applications can be found where autonomous agents can be trusted to automatically perform tasks on

behalf of the user. Given current agent technology, these domains should possess a moderate level of complexity and the penalty endured in case of failure should be low.

Manufacturability optimisation in feature based design is a potential domain where autonomous agents can be trusted to assist users. An agent driven system can be trusted to perform basic optimisation tasks on a design such as proximity or access problems, leaving the designer to concentrate on functional issues of his work.

2.2 Features as agents

The approach presented in this paper applies software agent technology to the feature level within the Feature Based CAD system. This radical move from traditional tried and tested architectures to an experimental agent-based architecture results in a new type of product model; a system where the designer creates a product model as an active community of agents rather than a passive data structure.

The feature level is chosen for "agentifying" as it best encapsulates the intrinsic manufacturing information. Lower level geometric entities such as points and edges contain little manufacturing information. The features implemented, such as holes, slots and pockets, have a simple correspondence with machining cycles on particular types of machines and so inherit the limitations of these machining cycles. Such feature agents maximise manufacturability through use of embedded knowledge of these limitations as well as template solutions to them.

Manufacturability of a component is largely determined through the geometric interaction between features in a design. Thin wall and tool accessibility problems arise from undesired geometric interactions inside the model. With feature agents, knowledge of required interaction states is embedded at a local (feature level) along with strategies to reach these desired configurations. This results in an active product model that capable of ensuring manufacturability through a global (component level) emergent behaviour.

3 Related Work

3.1 Feature-Based Design

3.1.1 Feature Recognition

The technique of feature recognition aims to extract individual form or machining features from a geometric representation of a component [**4, 5, 6**]. Indirectly it solves many manufacturability problems. Automatic recognition is usually very efficient for isolated features. However, there are limitations particularly the difficulty in appropriately recognising intersecting features. However, as new developments are made, the success rate of such systems improves, making them invaluable tools for process planning, reducing the manual work to recognition of only the most complex feature interactions.

3.1.2 Design by Features

At the other end of the spectrum are "design by features" systems [**4, 7, 8, 9**]. In such an environment, the designer creates a component by adding entities picked from a given set of basic form features and by applying regular geometric transformation to these entities. This approach, although constraining to the designer, avoids the pitfalls of feature recognition. In particular, complex interactions between form features pose no special problems to such systems, as each feature is, by design, a stand-alone entity. The design constraints so imposed help to ensure the manufacturability of the components created. Indeed, the supported features are usually linked to known machining operations or groups of operations on known machines. The geometric transforms are also selected to be compatible with these machining procedures. Thus, even before the design is started, each feature can be individually process planned and the only limitation on the manufacturability of a model comes from the geometric interactions between them. The downside is that all legacy designs must be recreated.

3.2 Manufacturability analysis

The automatic detection of potential manufacturing problems during the design stage can reduce time to market and save valuable resources by reducing tests on the shop floor to a minimum [**10**]. Indeed a reliable design validation tool could guarantee that a design sent to production is not returned for design faults relating to the manufacturing process. Such "quality control" on the design produced could greatly reduce the test phase and allow the process experts to concentrate on the more profitable process optimisation phase.

Automating the manufacturability analysis is often taken as part of the general process planning activity. Systems already exist that can assess a design, generate process plans and detect potential problem in a design. Such systems are surveyed in [**11**]. Although moderately successful, these systems remain limited in two important respects. Firstly,

in the type of geometric data they can process. Some will only accept purely 2½D geometry, others deal only with turning profiles. These geometric limitations do not seem to be a major handicap now as it is usually economically more viable to restrict the manufacture of a mechanical part to the minimum number of processes. The second major limitation of existing systems is their lack of initiative and solving capabilities. Some work has been done toward automatic generation of re-design suggestions [**12, 13**] but detection of problems is as far as most systems will go. Although such early detection is valuable, a tool that could solve a proportion of the manufacturability problems on behalf of the designer would be greatly beneficial.

3.3 Agents in Manufacturing

The application of software agents in the manufacturing context is a field of active research and development [**14**]. The majority of applications of multiagent technology in engineering design and manufacturing focuses on enterprise integration or manufacturing planning, scheduling and control. These systems chiefly uses coarse-grained agents as autonomous communicative entities encapsulating or replacing existing systems. These agents provide communication and knowledge sharing abilities to allow automatic data exchange between different design and production stages.

The GNOSIS research program [**15**] proposes a fine-grained approach that associates design features with autonomous agents. These feature-agents initiate dialog with other agents (machine tools, stock manager, etc.) in order to achieve greater concurrency in the product creation process.

4 Design Features

The feature-based system presented is in the "design by feature" category. It utilises a restricted set of form features to demonstrate the potential of feature agents. These form features are sub-divided into two categories. Positive features represent matter (the stock) while negative features define material removal (through machining). Figure 1 shows the features used in the system and their specific geometric parameters.

Blocks and cylinders are the two positive features available and are used as a base for all component designs. Holes, slots and pockets are the three negative primitives that the designer can use to "shape" a component. The position of a feature's origin, its orientation and a specific number of dimensions define the feature. Other attributes such as dimensional tolerances and surface finish can also be added for process planning, though they don't affect the nominal geometry or the geometric reasoning performed.

In order to reduce the complexity of the geometric algorithms in the prototype implementation, the modelling capabilities of the system are limited to strictly 2½D components. That is, the object can be machined on a 3-axis mill with the restriction that the feed axis (the ½ dimension, usually z) is never used simultaneously with the other two.

5 Manufacturability Criteria

Though examination of problems in typical components and a review of a number of manufacturability analysis systems have revealed many manufacturability criteria, the system described here implements four basic criteria relative to the milling and drilling process of prismatic components.

- **presence**

  Presence in the finished part requires that each feature should contribute to at least one surface of the finished part boundaries. This could be also called the geometric "usefulness" of a feature.

- **proximity**

  The close proximity of two features, one at least being negative, can generate thin walls of material that may be unable to withstand the stress of the cutting process. Such potential ruptures have grave consequences on the production process, requiring scrapping of both part and tooling.

- **collision**

  Collisions are geometric configurations of design features that are either physically impossible (intersection of positive features) or that could prevent machining of the part.

- **access**

  In order to be machined, a negative feature has to be accessible to a cutting tool. Access can be obtained either directly when the feature emerges to space (with appropriate orientation) or indirectly through another accessible feature. Access is of paramount importance and used to detect and resolve a large number of unmanufacturable designs.

These four basic criteria are inspired by the feature validation rules of [16]. Although providing an incomplete view on manufacturability, they permit the detection of a significant proportion of manufacturability problems. They can prevent the submission to the process planner of designs that are clearly not machinable or sub-optimal in design. The work on feature interaction by [17] suggests that other useful criteria could be added to the system. For example, Bidarra describes the splitting of the workpiece during the machining of a feature. This criteria could be could be used by features agents for autonomous detection and solving.

6 System architecture and operation

6.1 Communication language

The Knowledge Query and Manipulation Language (KQML) developed at the University of Maryland [18, 19] is a layered language, offering complete separation between communication data and message content. The domain of interest can be defined inside the content layer allowing heterogeneous applications to communicate meaningfully. For these reasons and the simplicity of parsing and generating KQML, it is fast becoming a *de facto* standard in the agent community. The system uses a subset of KQML as the shared language between agents.

6.2 System activity through Communication

Peer to peer communication inside the agent community is the source of all activity within the component. Two basic mechanisms are responsible for keeping the agent community alive and responsive at all times.

Firstly, whenever the internal geometry of a feature is modified (including creation and destruction), by the user or by itself, a feature agent automatically broadcasts a notification message to all other agents in the system. This *reflex* propagates any geometrical change to the entire system, ensuring that all agents have up-to-date information at all times.

Secondly, on reception of such a notification message, a feature agent systematically analyses the new geometry against itself and applies its embedded knowledge to detect and solve potential manufacturing problems. If a problem is detected during analysis that requires the feature to modify its own geometry, it will in turn broadcast this change to the rest of the agent community.

Thus, the activity of the system is maintained by the autonomous flow of geometric data between agents in the system. Propagation of changes through message broadcasting allows the system to responds immediately to any changes made by the user. Moreover, it allows a dormant system (in a stable mode) to automatically awaken in response to geometrical modifications.

6.3 BDI Approach

Among the various agent architectures available, the BDI (Beliefs-Desires-Intentions) scheme was chosen for the feature agents as it is most suited to support the set of criteria (desires) that represent a feature's manufacturability. The BDI architecture provides a mental state for each agent [20, 21, 22]. These mental states are separated into distinct categories:

- A **Belief** database

  Contains all the knowledge the agent accumulates during its life. The agent dynamically updates it to reflect as closely as possible the current situation of the agent in its environment. Some of these beliefs will be provided initially but most of them will be collected and revised by the agent during the course of its life. Agents generate two types of beliefs. *Global* beliefs reflect the manufacturability of the feature relative to its entire environment. They are implemented as counters:

```
PRESENCE        No of features granting presence on finished part
COLLISION       No of features colliding
PROXIMITY       No of features creating thin wall
ACCESS          No of possible tool access routes
```

  Individual beliefs are also generated that represent the relation between a feature and other features individually. They are implemented as Boolean variables. To avoid confusion between individual and global beliefs, all individual beliefs have a name starting with underscore ("_"). The individual beliefs supported by the system are the following:

```
_PRESENCE               feature is present in the finished part
_ABSENCE                feature is completely absent from the part
```

```
_ABSENCE_PARTIAL        feature is partially absent from the part
_COLLISION              feature collides with another feature
_PROXIMITY              feature is closer than pre-defined clearance
_ACCESS_ALLOW           feature has tool access
_ACCESS_DENY            feature has no tool access
_ACCESS_PARTIAL_ALLOW   feature has partial tool access
_ACCESS_PARTIAL_DENY    feature has partial access problem
```

Combining all individual beliefs contained inside the agent produces the counters used as global beliefs. The rules use to generate them are the following:

```
PROXIMITY = Σ(_PROXIMITY)
COLLISION = Σ(_COLLISION)
ACCESS = If (Σ(_ACCESS_ALLOW) > 0) Σ(_ACCESS_ALLOW)
         Else 1 - Σ(_ACCESS_DENY)
PRESENCE = If (Σ(_ABSENCE) > 0) 1 - Σ(_ABSENCE)
           Else Σ(_PRESENCE)
```

PROXIMITY and COLLISION are simple counters. ACCESS and PRESENCE are more complex. Indeed a single _ACCESS_ALLOW is enough to cancel any number of _ACCESS_DENY. Similarly, a single _ABSENCE cancels any number of _PRESENCE. It should be noted that partial contributions (_ACCESS_PARTIAL_ALLOW, _ACCESS_PARTIAL_DENY and _ABSENCE_PARTIAL) are checked prior to the application of these rules. For example if two partial accesses (_ACCESS_PARTIAL_ALLOW) combine to provide a full access route, Σ(_ACCESS_ALLOW) is incremented to reflect the situation.

- A **Desire** database

  Contains the set of facts that the agents "wants" to see realised. It represents a goal in life for the autonomous agent and is the driving force in the activity of the agent in the system. These desires can be dynamic and even dynamically modified by the agent, but in the current implementation, static desires are set at agent creation. Desires define acceptable values for beliefs from the manufacturing point of view. A positive feature representing a workpiece has the following desires.

```
PRESENCE > 0      ensures contribution to finished part
PROMIXITY = 0     no thin walls
COLLISION = 0     no collisions
```

A negative feature possesses desires that capture the limitations of the machining process.

```
PRESENCE > 0      ensures contribution to finished part
PROMIXITY = 0     no thin walls
COLLISION = 0     no collisions
ACCESS > 0        ensures tool access
```

- A **Plan** (or action course) database

  A database of potential action plans represents the "know-how" of the BDI agents. Plans are defined using a set of pre-conditions that are used to assess the applicability to given situation and post-conditions representing the expected outcome of executing that plan. Plans are strongly linked with both the desires and beliefs. Typically, a plan is chosen in order to realise the agent's desire and whose pre-conditions correspond to the current beliefs. The agent uses plans to generate intentions that lead to actions. Typically a plan is a solving routine dedicated to one manufacturability criteria that attempts to eliminate a desire/belief mismatch by modifying the feature's geometry or requesting another feature to modify its geometry. Several plans can exist for the same criteria that implement alternative solving strategies. For example, two strategies are implemented to solve access problems as show in Figure 5 (④ to ⑤). One is to translate the feature along z to the closest feature providing access, which preserves the feature's relative depth (distance between entry face and bottom face). The second is to also increase the depth so as to preserve the feature's absolute depth (z value of the bottom face). Although generating different actions, both strategies first perform a search of their surrounding for potential access routes. Known features are tested to find the closest negative feature (or union of features) granting access. If such a negative feature is found, the feature moves up ($+z$) to reach its bottom face. If not, the feature can still obtain access by modifying itself in order to emerge on the top face of the top-most positive feature currently denying access.

Figure 2 illustrates the internal operation of a BDI agent. The BDI engine endlessly compares the local desires against current beliefs. If a mismatch is detected, the plan database is searched for a suitable course of action. The choice is made according to the plan's pre-condition (matching the current beliefs) and post-conditions (solving a

belief/desire mismatch). Using this plan, intentions are generated and subsequently executed by the agent. Figure 3 shows the internal architecture of the feature agents. A KQML interface handles the complexity of communication between agents, including reception, filtering and posting of KQML messages. Received messages containing notification of geometric changes are routed to the main analysis routine. This routine contains all the geometric knowledge of the agent and extracts useful local information from the geometry received. Local beliefs (individual and global) are updated to reflect any new situation that has arisen from the new data collected. Running in parallel with the analysis routine is the BDI engine that generates the agent's course of action. Whether the action performed is a geometric transformation of the feature itself or a request to another agent, it always results in messages being sent to the rest of the agents in the model. This message may only be a notification of change in the geometry of the feature.

At all times, the BDI agent can receive data from its environment and analyse it to update its internal beliefs database. In parallel and autonomously, the BDI monitors changes in the belief database. When such changes occur, the engine compares the agent's desires against the current beliefs. If a mismatch is detected, the plan database is searched for a suitable set of actions. The BDI approach allows agents to always be in phase with their environment by strongly linking their action with the present situation.

6.4 Duality of Agent Activity

Figure 3 also shows the duality contained in each agent. One part of the agent is dedicated to geometric analysis while the other deals with generating behaviour. This separation is necessary to fully benefit from the BDI architecture. Indeed, by separating these two activities, it is possible to ensure that geometric knowledge about the environment is always up to date and therefore prevent course of actions to be taken based on outdated beliefs.

6.4.1 Geometric Analysis

The main analysis routine of the feature agents analyses the geometric data received. It extracts useful information by comparing the agent's own geometry with that received. Properties such as intersection type, minimum distance or minimum angle are calculated and stored in the local belief database so avoiding the repetition of identical tests. These stored beliefs are then used in the geometric analysis to update the static beliefs corresponding to the agent's desires (presence, access, collision, and proximity).

Dynamic local beliefs in the agent's database are classified by feature to allow fast retrieval and updating. Specific beliefs about other feature agents are combined into global beliefs that give an overview of the type of inter-relationship. These beliefs are stored along with a reference to the agent concerned. This storage scheme permits a two-way retrieval of beliefs and agents concerned.

6.4.2 Behaviour Generation

The behaviour generation takes place inside the BDI engine and runs in parallel with the geometric analysis. The results of this analysis determine if and how it should act.

The BDI engine probes the belief database at regular intervals and checks for any mismatch between the agent's desires and the current beliefs. If one or more mismatches are detected, a plan that fits the current situation is applied to try to resolve the existing manufacturing problem. If no suitable plan can be found in the current situation, the BDI engine is put in stand-by mode until the geometric agent collects new data. This prevents the plan database from being searched repeatedly in a situation where no suitable plan exists.

A local belief update is enforced after any local geometric change resulting from the application of a plan. A broadcast of the new geometric parameters of the feature is also performed to permit the change to propagate throughout the model.

7 Service Agents

Feature agents are lightweight computing entities that possess only simple analytic power. Their strength comes from their ability to organise into a community. However, various tasks related to design or CAD system operation need to be performed that are not supported by feature agents. These tasks are performed by service agents. Service agents are middleweight or even heavyweight computing entities that provide high level services to the community of feature agents.

Services such as screen display, inter-agent constraint management, inter-agent communication management or even finite element analysis are service agents that enrich system functionality.

8 Experimental Implementation

An experimental implementation of the principles described has been successfully completed. Two limitations can be noted right away. The system remains comparatively limited in its geometric modelling capabilities and perhaps has too much trust in the ability of the features agents to modify the design. However, it demonstrates the validity of the concepts and offers a glimpse of the potential application of feature agents as a powerful tool to aid mechanical design.

8.1 Swarm Agents

The multi-agent engine used in the prototype implementation is the Swarm [23] engine developed at Santa Fe Institute. Swarm's target audience is the artificial life community and is thereby more adapted for simulation runs than powering a dynamic CAD application. However, it is both powerful and flexible enough to accommodate the approach described in this paper.

8.1.1 Self scheduling

The main obstacle to the use of Swarm to implement autonomous agents was the schedule-based nature of its multiagent mechanism. This was overcome by creating a new type of self-scheduling agent that controls the central schedule rather than relying on the Swarm clock mechanism.

Swarm simulates multiagent concurrent activity with a system of activity schedules. A schedule consists of a clock and a mechanism to plan and perform actions inside an agent community (a swarm) according to the value of the clock. Simple Swarm agents rely on fixed periodic schedules allowing them to perform the same actions for each schedule period. A typical cycle of a schedule could be data collection, data analysis, and action. This cycle is repeated at each schedule cycle for every agent in the system. It is an efficient approach to simulate concurrent activity in simple cases (though it can be resource consuming) and allows agents to use their initiative.

The creation of self-scheduling agents solved these limitations. That is, agents that can autonomously add or remove items in their schedule at runtime. The community of feature agents uses a non-periodic dynamic schedule to drive its activity. Each agent can access the community schedule, book available time slots and add items to it to perform its activity. This mode of operation scheduling is very similar to that used by most multitasking operating systems and gives the illusion of parallel processing.

8.1.2 Coupled agents

Previously, the dual activity (geometric and behavioural) taking place inside each feature agent was described. All the dynamic beliefs about geometric data are stored inside the geometric agent while the static desires used for reference are kept inside the behaviour agent. This separation allows the geometric agent to work independently from the behaviour agent. A consequence of this modularity is that the behaviour generation of a feature can be switched on or off at runtime independently of the geometric analysis. This allows a feature to maintain up-to-date beliefs even when behavioural responses are not required.

The geometric and the behaviour agents perform their auto-scheduling in separate schedules, further increasing their separation. Swarm guarantees the synchronisation of these two schedules, by transparently collapsing all sub-schedules into a single, system-level schedule at runtime.

8.1.3 Service Agents

Some service agents have been implemented and added to the system to provide high-level functions not performed by the feature agents.

- A *display agent* collects geometric data from the model, translates it into a suitable representation and sends it to a 3D package for on-screen visualisation.
- A *constraint manager agent* handles basic inter-feature constraints such as concentricity, relative orientation and position. It is able to check changes in the model geometry against a dynamic list of constraints. When a constraint is violated, the agent can attempt to find a solution and realise it by sending requests to feature agents.
- An *activity monitor agent* keeps an eye on the global activity of the system, detects livelock situations and halts all activity when they occur.

8.2 Global architecture

Figure 7 illustrates the global architecture of the prototype system. As is common in a Swarm application, the system's activity is supported by two separate swarms.

The *model swarm* contains all feature and service agents that create the dynamic product model. It also holds a KQML facilitator, which behaves as a central post-office for KQML messaging and a user agent, which is an empty agent shell allowing the designer to emit requests to features in the same manner as another agent. The *observer swarm* exists to interface the model with the outside world (user and other applications). In particular, it handles the creation of new features and their introduction to the model swarm.

8.3 Results and analysis

An experimental implementation of the feature-based, agent-driven design system has been realised. This working system uses the Swarm libraries to power the agents and allows creation of 2½D components using a set of five design features. The core of the system is coded in ObjectiveC using the Swarm libraries but a Dynamic Data Exchange (DDE) link to the ACIS 3D Toolkit [**24**] is implemented to provide visual feedback to the designer. The ACIS toolkit can also be used to export the obtained designs as pure 3D geometry.

The resulting system offers a demonstration of the use of autonomous agents in feature based system. The multiagent system offers autonomous real-time analysis of the design and can even perform geometric corrections on behalf of the user.

Important characteristics of the multiagent system are described in more detail before discussing the approach advantages and shortcomings.

8.3.1 Global emergent behaviour

The experimental implementation of the system demonstrates the principle of emergent behaviour in a multiagent structure. The entire interactions within the system are modelled at agent level. The only code written at the application level is to initialise the Swarm multiagent engine.

Despite the absence of any top-level code for synchronisation or organisation, the system behaves in an ordered fashion throughout the ensuing agent interactions. For example, no global (application level) processing is necessary to propagate geometric changes throughout the model. Instead, each agent is responsible for propagating local changes to the rest of the community. Similarly, the collection of geometric data is performed locally by each feature without global intervention.

8.3.2 Design example

Figure 4 shows a simple bracket design that is used to illustrate the internal state of feature agents inside a component. Each feature agent inside the system maintains a local belief database (see Figure 3) reflecting its relationship with its environment. Beliefs are divided into two categories, *individual* and *global.* Consider the feature "counter hole left" and "hole left" of the bracket component. Their belief databases are given in Table 1 and Table 2.

Notice that the feature "block" provides "counter hole left" with presence but also completely denies tool access to it. However, "slot left" provides a full access route. Globally "counter hole left" fulfils all its desires (see 6.3) with a presence and access of 1 and no collision or proximity problems.

Like "counter hole left", "hole left" is denied tool access by "block". However, it obtains a viable access route through "counter hole left", which allows it to fulfil its desires. Interestingly, it must be noted that "hole left" is only concerned about its local accessibility. That is to say, it does not check that "counter hole left" is also accessible before considering its ACCESS desire fulfilled. An implicit delegation of tasks exist between these features and it remains the responsibility of "counter hole left" to ensure its local accessibility. The bracket is a simple design intended for demonstration but the system scales well with more complex designs (see Figure 6).

8.3.3 Self correcting model

The emerging function of the active product model is to perform self-correction on the design in order to guarantee global manufacturability of the proposed design. These corrections occur autonomously without intervention of the user.

Figure 5 shows cases of autonomous corrections of a model during a typical design session. For the sake of simplicity the model used in this design session is the bracket previously detailed and illustrated in Figure 4. Five

significant stages of the 2½D model are shown along with corresponding cross-sections. The central arrow illustrates the session's progression and summarises the important actions taken by both designer and feature agents.

i. In ①, the session starts with a stable design (without the centre slot of Figure 4). All the features are in a "satisfied" state, so system activity is minimal.

ii. The user decides to add a centre slot as reflected in ②. The slot is created and immediately broadcasts its new geometry throughout the model.

iii. With their local beliefs updated, "slot centre" and "block" detect a proximity problem, as they cause a thin wall section at the edge of "block". "Slot centre" applies a default avoidance strategy to solve the proximity problem. This results in the slot changing its position (as shown in ③) and the model reaching a new "satisfied" state.

iv. In order to save machining time, the user decides to try the design without the counter holes. The two features are deleted triggering a notification broadcast. Two access problems are now detected in ④.

v. "Hole left" and "hole right" apply alternate plans to solve their access problem and reach stable state ⑤. This difference reflects preferences previously expressed by the user. On the one hand, "hole right" ensures its absolute depth by translating itself along z in order to obtain access through "slot right". On the other hand, "hole left" guarantees an absolute bottom z value by changing its depth and position in order to gain access through "slot left".

The designer decides at runtime on the extent and type of self-correction performed by the model. Agents can be constrained by switching off one or more of their individual manufacturability criteria. In addition, as demonstrated by "hole left" and "hole right" in this example, features can use different basic behaviours in order to suit the designers needs. These preferred behaviours are accessible to the user at any time during a design session.

Unlike traditional CAD packages, when using the agent-driven feature-based system the user does not have to wait for a complete parametric update after each model correction. He can continue to modify a design as the agents perform their tasks.

8.3.4 Autonomy vs. Stability

The particular issue of model stability arises from the agent driven approach. Indeed, allowing features a high degree of autonomy inside the design can help self-correcting of the design but can also lead to a loss of partial designs. A balance between feature autonomy and design stability must be struck for the system to be of value.

Agents can be individually motivated by different manufacturability criteria. It is possible for the designer to enable and disable individual criteria at runtime. It is also possible to select different behaviours for each criterion as described in Figure 5 (transition from ④ to ⑤). This allows the level of autonomy of individual agents to be altered dynamically within the active model. For instance, the designer can choose to disable all behaviours for parts of a design that are already satisfactory. This will freeze all the features involved in their current positions while allowing others to remain active and self-correcting. It is important to note that although the behaviours can be switched off, this is not true of the geometric analysis. This allows partially active models to work since even "disabled" agents can provide the geometric information needed for undertaking behaviours.

8.3.5 Conflict resolution

Conflicts between feature agents might arise during the course of operation. It is important that such conflicts be handled gracefully by the system. Two main problems can occur when such conflicts arise.

- A *deadlock* is a situation where two or more agents inside the system can no longer perform their task because they rely on unavailable results. Typically, agent A waits for agent B before performing a task. If agent B is itself relying on agent A's result, neither can function and the system is deadlocked.

- A *livelock* is a related phenomenon identified in multiagent systems. It occurs when an action performed by agents induces cascading cyclical responses from other agents in the system. Livelocks results from cyclical chains of events within the agent community. Such a chain of events leads the system to endlessly encounter the same problem as its attempts at a solution merely creates a similar problem.

The system gracefully handles conflicts. Provided no classical programming deadlock (infinite loops, cross-locked memory access etc.) exist within the agent's code, deadlocks can never arise, as a feature agent does not rely on any external data other than that volunteered by its peers, analysed and then stored in its local belief database. Thus, each feature agent in the system is completely independent from its peers' activity when action is taken despite relying on the data collected from the rest of the agent community. The value of the system therefore depends on the volunteering of geometric data between agents rather than on compliance to information requests. The ability to

guarantee no inter-agent deadlocks is of critical importance as such a deadlock can completely and permanently halt the agents' activity and deactivate the entire model.

On the other hand, the avoidance of livelocks cannot be guaranteed within the current implementation. Indeed it is possible to find particular geometric and behaviour (design criteria enabled/disabled) configurations that could lead to livelocks. It is important to note however, that in the case of CAD, livelocks usually do not threaten the global utility of the system. It is always possible for the user to force a solution to a conflict and regain an operational model. Yet, the hyperactivity that result from livelocks can threaten the stability of the model by propagating unnecessary modifications to a stable part of a design. This motivated the addition of the activity monitor agent to the prototype. This service agent automatically detects potential livelock and subsequently pauses all activity in the model before partial solutions are lost through hyperactivity.

8.3.6 Graceful degradation

Sometimes, feature agents can be confronted with problems they can not solve That is to say, situations where the plan search does not yield any suitable course of action. In such a situation, the system should not lock itself or go into an activity overdrive. It should gracefully degrade and remain operational.

The current architecture allows for such graceful degradation. If a geometric configuration arises where manufacturability is compromised but no suitable plan can be found, a feature agent does not perform any actions and does not send any broadcast notification. Instead, it just returns to default activity, maintaining local beliefs and assessing manufacturability after each change in beliefs. This inactivity has two beneficial side effects. Firstly, because the BDI engine acts upon changes in beliefs, it prevents unnecessary failed searches of the plan database. Secondly, the graceful "inactivity" of one agent might give another involved a chance to solve the problem, increasing the flexibility and adaptability of the system.

8.3.7 Computer Aided Process Planning (CAPP) pre-processing

The multi-agent community representing the model performs many geometric tests during the course of the design and stores the results inside each agent's local belief database. It is therefore able to provide much more information about the model than pure geometry by volunteering these partial results. In particular, each feature agent is able to provide useful data concerning local accessibility to a CAPP system. Indeed, local beliefs concerning indirect access (through one or more other features) can be easily expressed as precedence constraints in the machining sequence. CAPP system usually determines these constraints and can therefore be simplified and focus on other planning problems such as machine and tool selection and scheduling.

9 Advantages and Drawbacks of the system

The approach described in this paper demonstrates that a major transformation could be achieved in feature-based CAD packages by means of a multiagent implementation at feature level. Through autonomous agents it is possible to place a large part of the design activity inside the model itself. The active model created can be trusted to conduct manufacturability testing and optimisation on behalf of the user and in real time. A self-correcting model is obtained by making individual design features partly responsible for their own geometry. It performs repetitive tasks, leaving the designer free to concentrate on more interesting aspects of the design as well as ensuring maximum manufacturability of the design. The described model activity remains under the ultimate control of the designer, who can choose between different agent behaviour or even disable individual activity all together.

The radical change towards an active product model also creates some new problems that need to be addressed. Deadlocks and livelocks are the two most obvious problems common to most multiagent systems and can be tackled using results from the multi-agent research community in the area of co-ordination and negotiation [**25, 26**]. The problem of model stability that arises from the creation of an active product model was shown in section 8.3.4 to be efficiently addressed using flexible behaviour management inside each feature agent but could benefit for additional research work. Another important issue concerns model storage and loss of knowledge. Traditional file formats for mechanical parts (e.g., STEP, IGES) are not well suited for saving the living community that the product model has become. Using them to store the model results in substantial loss of knowledge. Indeed, both behavioural preferences and local beliefs held by feature agents would be lost if such geometry-oriented formats were used. Agent serialisation (in the OO programming sense) is an easy solution for storing models in files at the cost of data redundancy. Finally, it can be noted that the solutions offered by the system are partially dependent on the order of introduction of features into the model. Again, flexible behaviour management can help minimising the impact of this dependence by allowing "batch" introduction of disabled agents.

10 Future work

Communication load within the system can become very heavy in large models. Indeed, with notification broadcasts fired after each geometrical change, the communication grows exponentially with the number of features. A new service agent is planned that will perform space partitioning in order to localise communication and thereby reduce the number of messages sent for each broadcast. Because feature agents perform based on their knowledge of their immediate surrounding, partitioning will not affect their efficiency in ensuring local manufacturability.

Livelocks remain a possibility in the system, and although they do not threaten the stability of the system, they can disturb the stability of the model being designed. It is clear that attempting to explicitly suppress livelocks through inter-agent negotiation would result in heavy complexity overhead. Instead, the activity monitor agent that detects potential livelocks provides a good compromise between simplicity and functionality. However, the activity monitoring can sometimes give false alarms in very active models and stopping the global activity remains a crude way of dealing with livelocks. A more subtle activity monitor is planned that would not freeze the entire model activity but just turn off specific manufacturing criteria of the features involved before prompting for the user's intervention.

It is evident that the system's potential modelling power could be greatly improved by providing more powerful inter-feature geometric relationships (or constraints). These relationships could be used to offer a better degree of support for dimensions and tolerances inside the feature-based models [**27**]. They could also offer a framework for propagating changes through part assemblies.

Extending the concept of CAPP pre-processing, work is under way to link Edinburgh's agent-driven design system to a holonic manufacturing system developed at the University of Nancy [**28**]. This will demonstrate that a properly validated model can be transferred directly from the design environment into suitable manufacturing holons for production.

11 Conclusion

A prototype mechanical design-for-manufacturability system has been presented that uses autonomous agents to implement design features. This major change in the system architecture creates a new type of approach to the design process and the nature of the model produced. The traditional monolithic CAD application is no longer at the centre of the design phase. Instead, the product data model is transformed into a product program model. The model itself is now the source of activity in the system. The autonomous agents inhabiting the model continuously test themselves against a set of manufacturability criteria and modify themselves to comply. The combined localised effort of all the feature agents results in a global emergent behaviour validating and ensuring part manufacturability.

Although lacking important engineering functionality, the experimental feature-based agent-driven system shows that properly motivated autonomous agents can be trusted to validate mechanical components and even to perform automatic corrections on behalf of the designer.

References

1 **Nwana, H. S., Ndumu, D. T.**, An Introduction to Agent Technology, Software agents and soft computing. *Lecture Notes in Artificial Intelligence*, November 1998, (Springer Verlag, 1998).

2 **Wooldridge, M., Jennings, N.R.**, Intelligent agent: Theory and Practice. *Knowledge Engineering Review*, **12**(2), pp. 115-152, 1995.

3 **Foner, L.N.**, What's an agent, anyway? A sociological case study. *MIT Media lab*, 1993.

4 **Salomons, O.W., Van Outen, F.J.A.M., Kals, H.I.J.**, Review of Research in Feature Based Design. *Journal of Manufacturing Systems*, **12**(2), pp. 113-132, 1993.

5 **Laako, T., Mantyla, M.**, Feature Modelling by Incremental Feature Recognition. *Computer Aided Design Journal*, **25**(8), pp. 479-492, August 1993.

6 **Little, G., Tuttle, R., Clark, D.E.R., Corney, J.R.**, The Heriot-Watt Feature Finder: CIE97 Results. *Computer Aided Design Journal*, **30**(13), 1999.

7 **Mill, F.G., Naish, J.C., Salmon, J.C.**, Design for Machining with a Simultaneous Engineering Workstation. *Computer Aided Design Journal* **26**(7), pp. 521-527, 1994.

8 **Salomons, O.**, Computer Support in the Design of Mechanical Product, *PhD thesis*, University of Twente, Enschede, Netherlands, 1995.

9 **Shah, J.J., Mantyla, M.**, Parametric and Feature-Based CAD/CAM: concepts, techniques and applications, Wiley & Sons, 1995.

10 **Gupta, S.K., Nau, D.S.**, A systematic approach for analysing the manufacturability of machined parts. *Computer Aided Design Journal*, **27**(5), pp. 343-352, 1995.

11 **Gupta, S.K., Regli, W.C., Das, D.** and **Nau, D.S.**, Automated Manufacturability Analysis: A Survey. *Research in Engineering Design*, **9**, pp. 168-190, 1997.

12 **Ong, S.K., Nee, A.Y.C.**, Manufacturability Evaluation and Generation of Re-Design Suggestions for Machined Parts, *The International Journal for Manufacturing Science & Production*, **1**(2), 1998.

13 **Das, D., Gupta, S.K., Nau, D.S.**, Generating redesign suggestions to reduce setup cost: a step towards automated redesign, *Computer Aided Design Journal*, **28**(10), pp. 763-782, 1996.

14 **Shen, W., Norrie, D.H.**, Agent-Based Systems for Intelligent Manufacturing: A State-of-the-Art Survey. *Knowledge and Information Systems: an International Journal*, 1999.

15 **Balasubramanian, S., Norrie, D.H.**, A multi-agent intelligent design system integrating manufacturing and floor-shop control, *ICMAS 95. Proceedings of the 1st Int. Conf. On Multi-agent systems*, Menlo Park, CA, USA, June 1995

16 **Vandenbrande, J.H., Requicha, A.A.G.**, Spatial Reasoning for the Automatic Recognition of machinable features in solid models. *IEEE Transaction on pattern analysis and machine intelligence*, **15**(12), pp. 1269-1285, December 1993.

17 **Bidarra, R., Dohmen, M., Bronsvoort, W.F.**, Automatic Detection of Interactions in Feature Models, *ASME Design Engineering Technical Conferences*, Sacramento, California 14-17 September 1997.

18 **Finin, T., Fritzson, R.**, KQML -a language and protocol for knowledge and information exchange. *Technical Report* [CS 94-02], (Computer Science department, University of Maryland, UMBC, Baltimore 1994).

19 **Finnin, T., Labrou, Y., Mayfield, J.**, KQML as an agent communication language. In *Software Agents* (ed. Bradshaw, J), (MIT Press, Cambridge, 1997)

20 **Fisher, K., Muller, J.P., Pishel, M.**, A pragmatic BDI Architecture, *Intelligent Agents II*, 1995.

21 **Norman, T.J., Long, D.**, Goal creation in motivated agents, *Intelligent Agents*, 1994.

22 **Rao, A.S., Georgeff, M.P.**, BDI-agents: from theory to practice, *Proceedings of the first international conference on multiagent systems*, San Francisco, 1995.

23 **Burkhart , R.**, Schedules of Activity in the Swarm Simulation System, *OOPSLA '97 Workshop on OO Behavioral Semantics*, Atlanta, Georgia, October 1997.

24 **Corney, J.**, 3D modeling using the ACIS kernel and toolkit, (Wiley 1997).

25 **Gaines, B.R., Norrie, D.H.**, Coordinating Societies of Research Agents - IMS Experience, *Integrated Computer-Aided Engineering*, **4**(3), pp. 179-190, 1997.

26 **Polat, F., Shekhar, S., Guvenir H.A.**, A negotiation platform for cooperating multi-agent systems, *Concurrent Engineering: Research and Applications*, **1**, pp. 179-187, 1993.

27 **Shah, J.J., Yan, Y., Zhang, B.-C.**, Dimension and tolerance modeling and transformations in feature based design and manufacturing, *Journal of Intelligent manufacturing*, **9**, pp. 474-488, 1998

28 **Chaxel, F., Bajic, E., Richard, J.**, Mobile Database Nodes for Manufacturing Information Management: a STEP Approach, *International Journal of Advanced Manufacturing Technology*, **13**, pp. 125-133, 1997.

Acknowledgements

The authors would like to acknowledge the Division of Engineering of The University of Edinburgh for funding this work through a PhD studentship.

Table of Tables

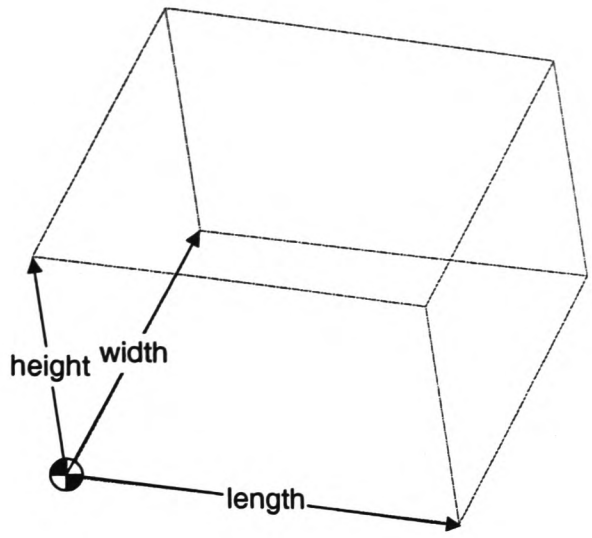
Table 1 "Counter Hole Left" Beliefs

"Counter Hole Left" *Local* Beliefs		(Boolean values: _TRUE and ~~_FALSE~~)	
About "Block"	About "Slot Left"	About "Slot Center"	etc …
_PRESENCE	~~_PRESENCE~~	~~_PRESENCE~~	
~~_ABSENCE~~	~~_ABSENCE~~	~~_ABSENCE~~	
~~_ABSENCE_PARTIAL~~	~~_ABSENCE_PARTIAL~~	~~_ABSENCE_PARTIAL~~	
~~_COLLISION~~	~~_COLLISION~~	~~_COLLISION~~	
~~_PROXIMITY~~	~~_PROXIMITY~~	~~_PROXIMITY~~	
~~_ACCESS_ALLOW~~	_ACCESS_ALLOW	~~_ACCESS_ALLOW~~	
_ACCESS_DENY	~~_ACCESS_DENY~~	~~_ACCESS_DENY~~	
~~_ACCESS_PARTIAL_ALLOW~~	~~_ACCESS_PARTIAL_ALLOW~~	~~_ACCESS_PARTIAL_ALLOW~~	
~~_ACCESS_PARTIAL_DENY~~	~~_ACCESS_PARTIAL_DENY~~	~~_ACCESS_PARTIAL_DENY~~	
"Counter Hole Left" *Global* Beliefs			
PRESENCE = 1			
COLLISION = 0			
PROXIMITY = 0			
ACCESS = 1			

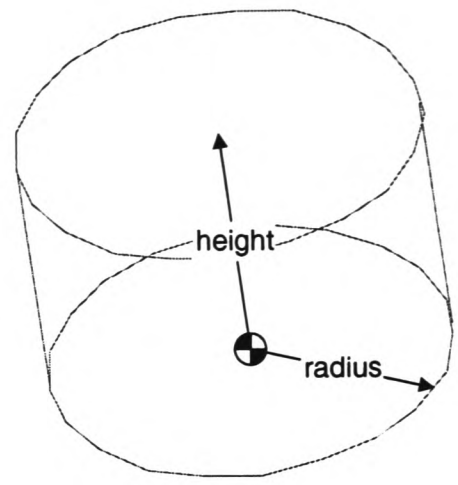
Table 2 "Hole Left" Beliefs

"Hole Left" *Local* Beliefs			(Boolean values: _TRUE and ~~_FALSE~~)
About "Block"	About "Slot Left"	About "Counter Hole Left "	etc …
_PRESENCE ~~_ABSENCE~~ ~~_ABSENCE_PARTIAL~~ ~~_COLLISION~~ ~~_PROXIMITY~~ ~~_ACCESS_ALLOW~~ _ACCESS_DENY ~~_ACCESS_PARTIAL_ALLOW~~ ~~_ACCESS_PARTIAL_DENY~~	~~_PRESENCE~~ ~~_ABSENCE~~ ~~_ABSENCE_PARTIAL~~ ~~_COLLISION~~ ~~_PROXIMITY~~ ~~_ACCESS_ALLOW~~ ~~_ACCESS_DENY~~ ~~_ACCESS_PARTIAL_ALLOW~~ ~~_ACCESS_PARTIAL_DENY~~	~~_PRESENCE~~ ~~_ABSENCE~~ ~~_ABSENCE_PARTIAL~~ ~~_COLLISION~~ ~~_PROXIMITY~~ _ACCESS_ALLOW ~~_ACCESS_DENY~~ ~~_ACCESS_PARTIAL_ALLOW~~ ~~_ACCESS_PARTIAL_DENY~~	
"Hole Left" *Global* Beliefs			
PRESENCE = 1 COLLISION = 0 PROXIMITY = 0 ACCESS = 1			

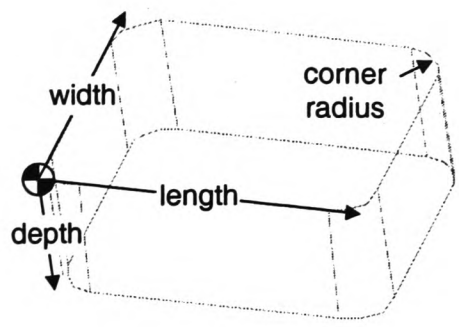
Table of Figures



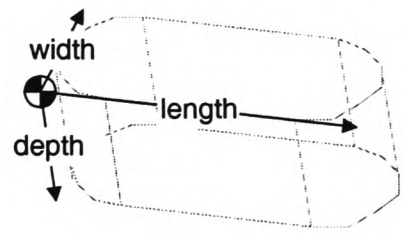
Block



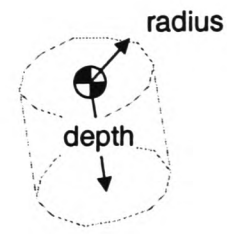
Cylinder



Pocket



Slot



Hole

Figure 1 Supported Design Features

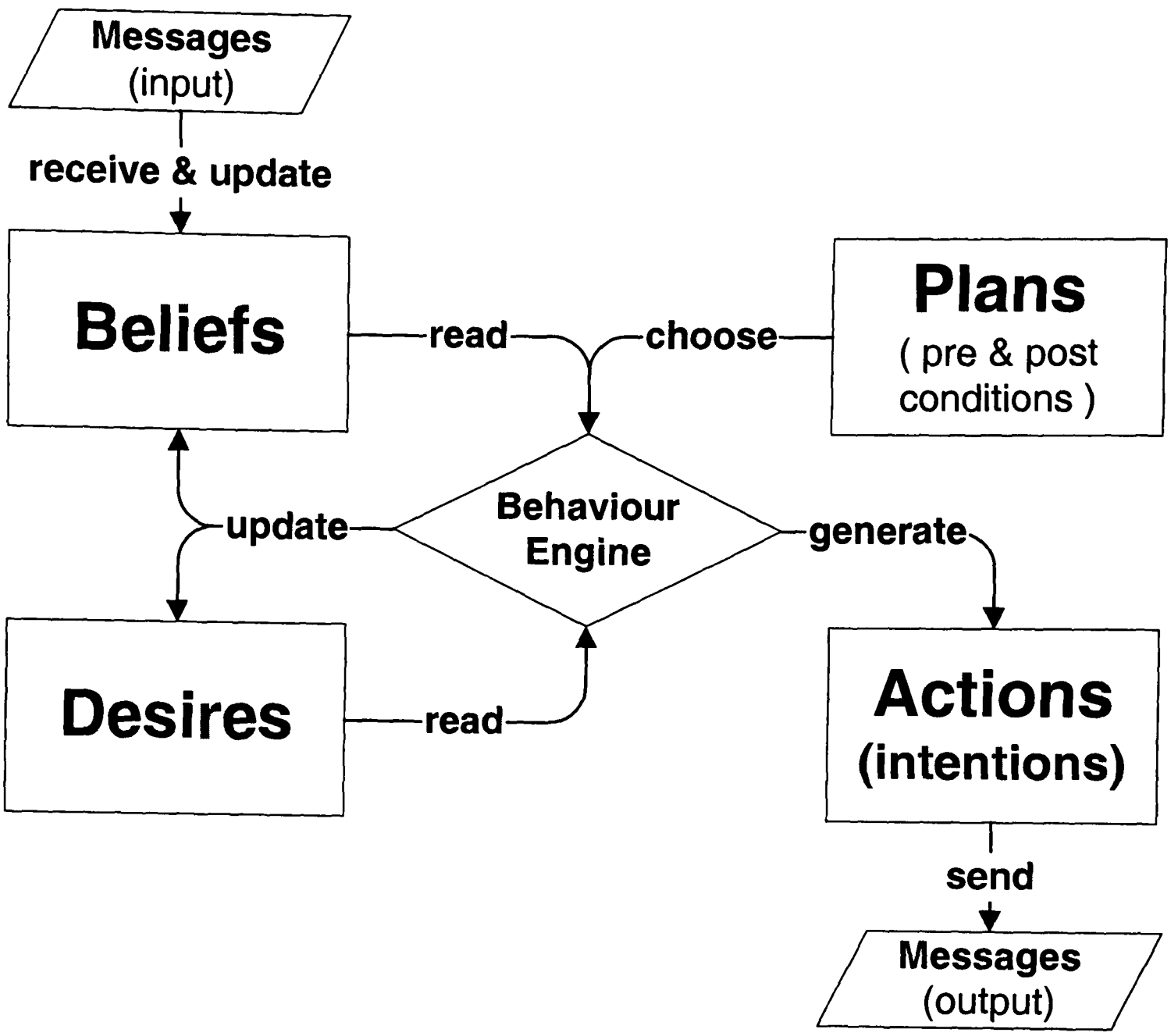


Figure 2 Internal BDI Operation

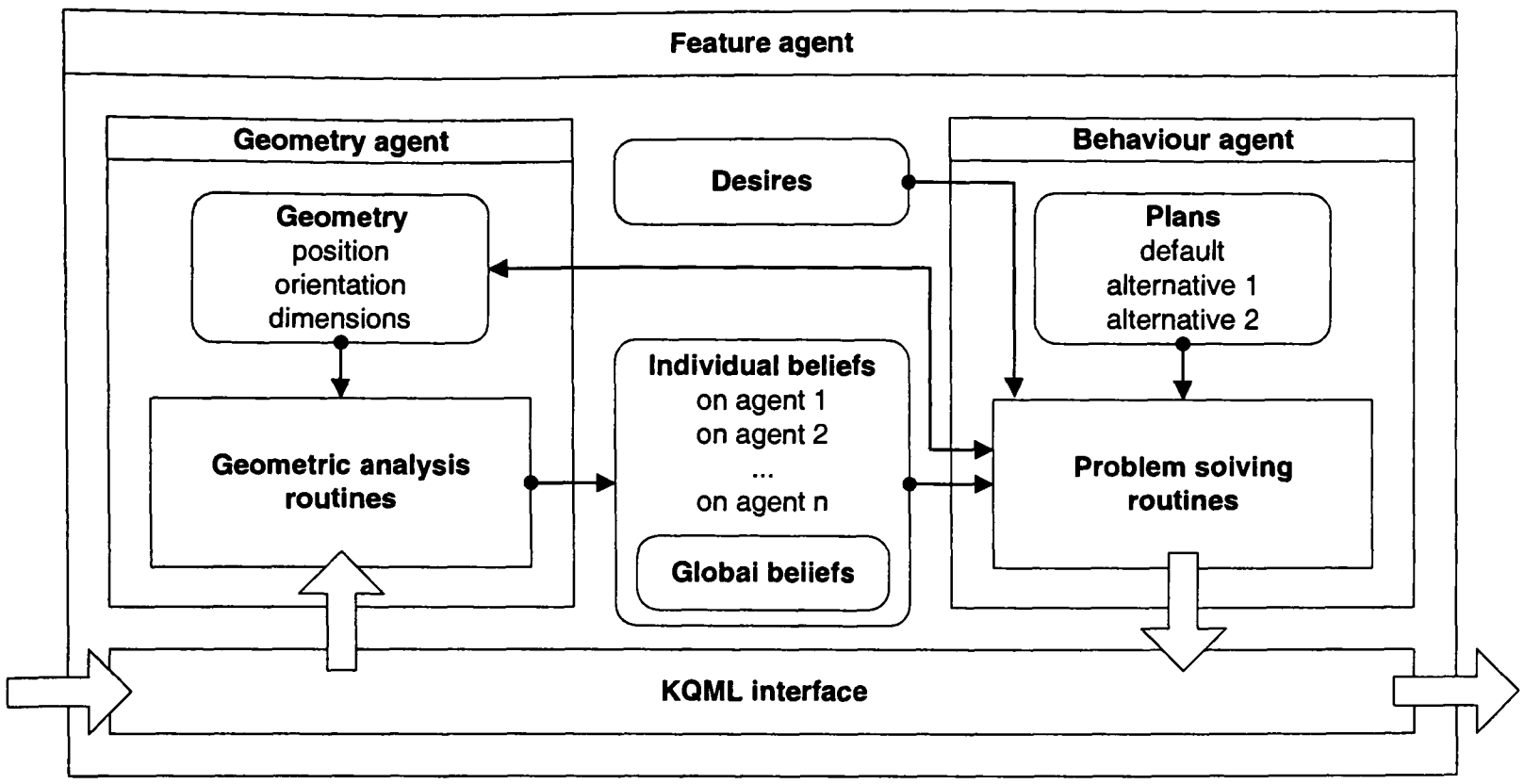


Figure 3 Internal Agent Architecture

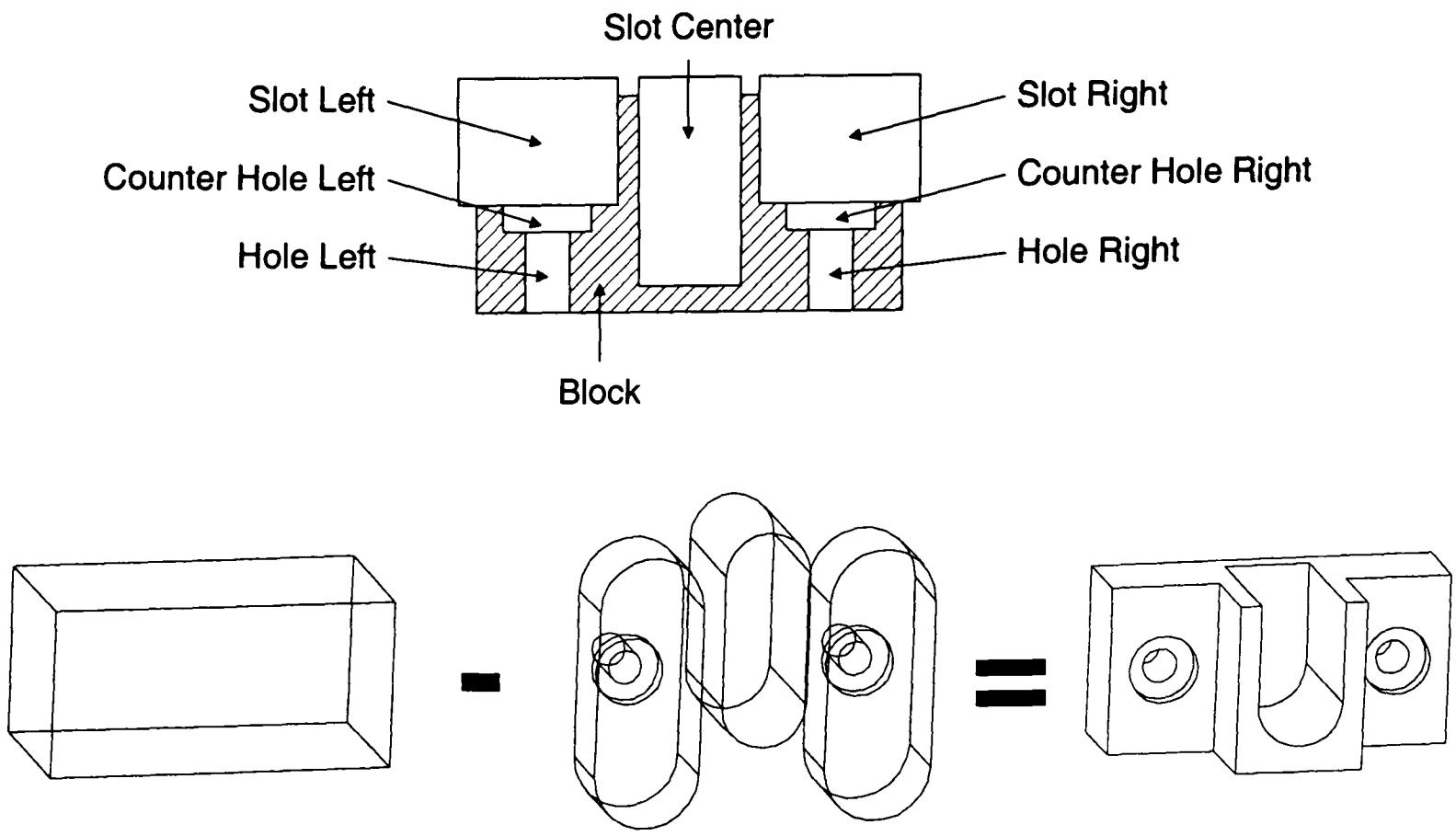


Figure 4 Bracket Design Example

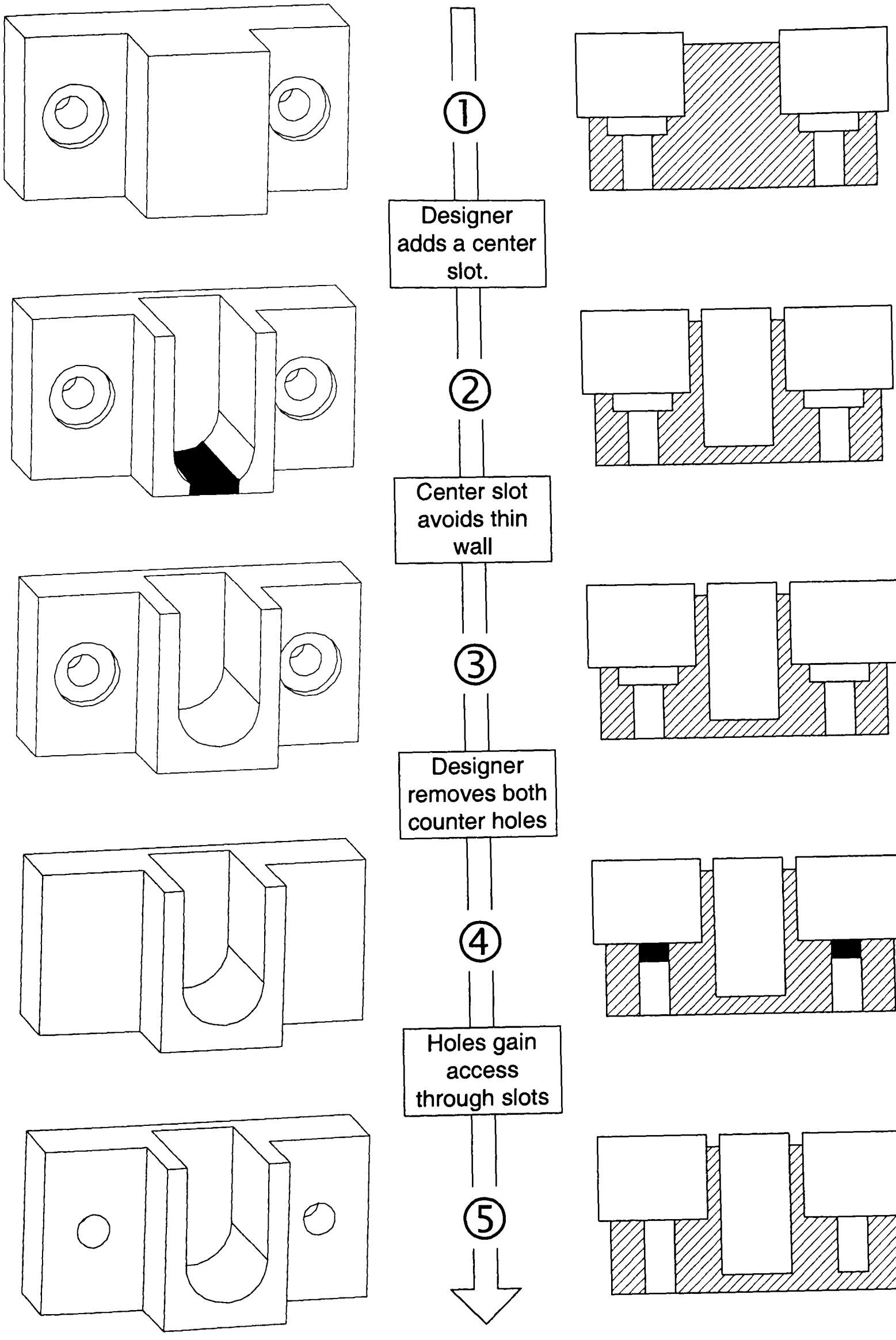


Figure 5 Self Correction Example

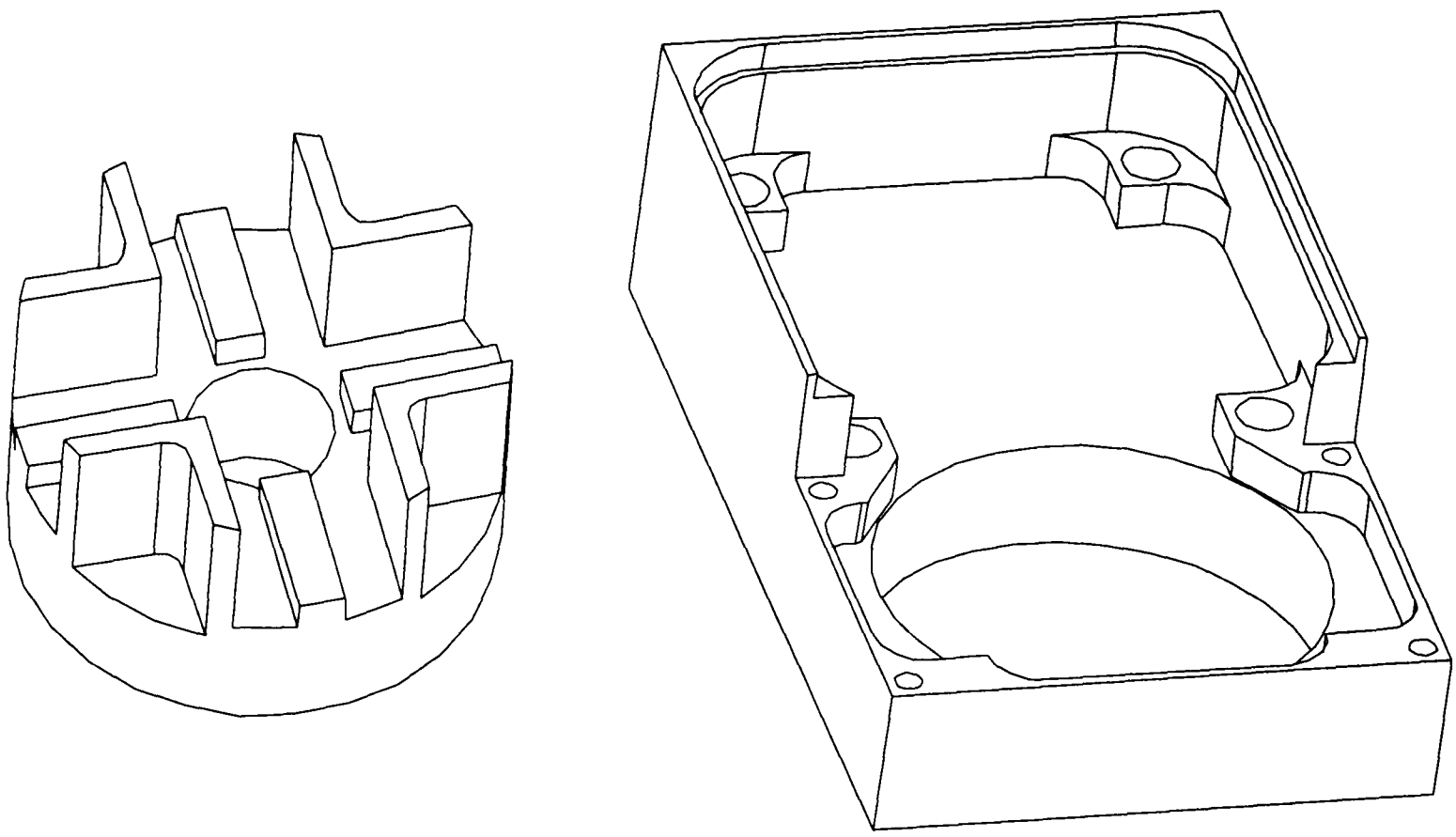


Figure 6 Examples of Successfully Modelled Parts

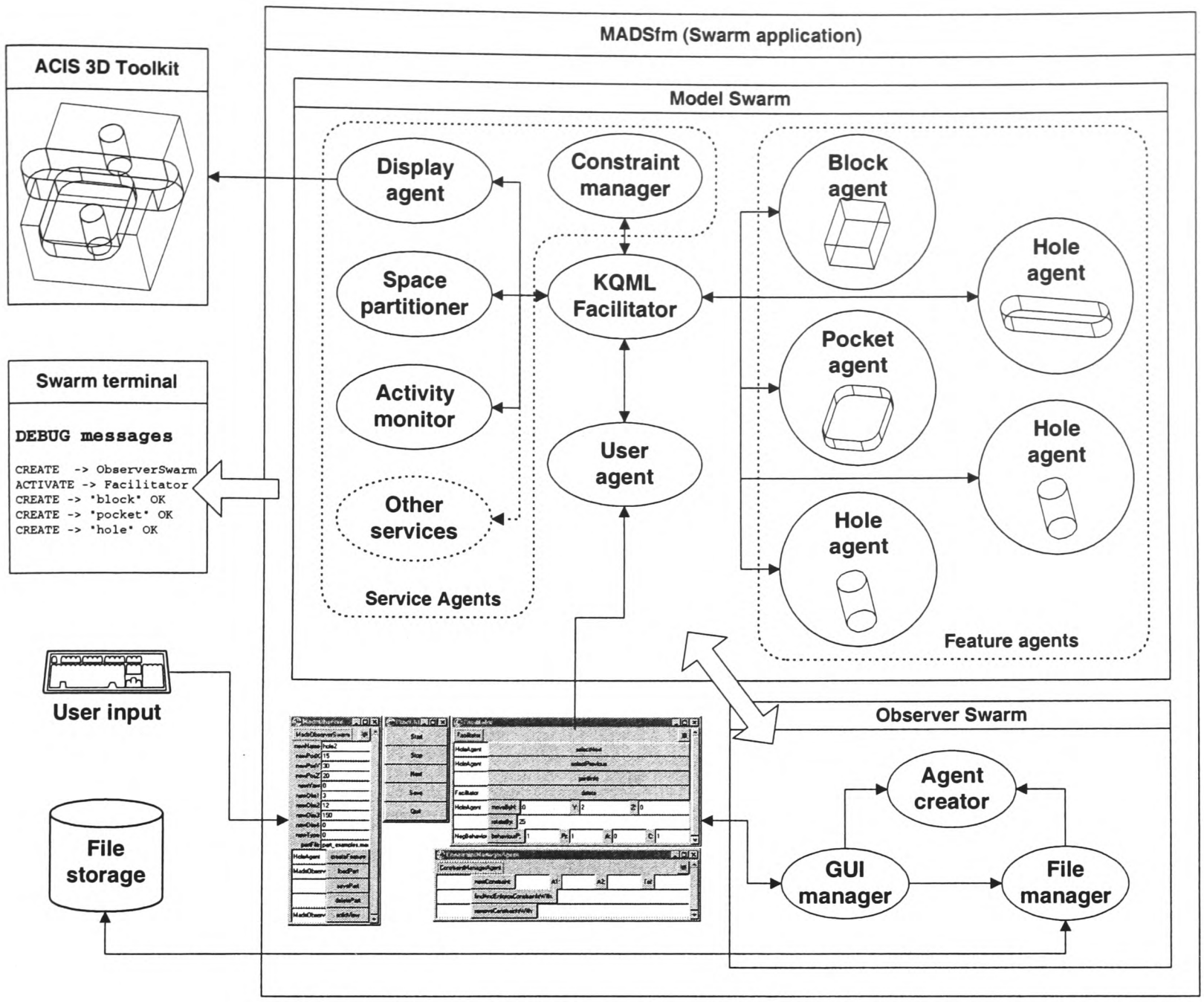


Figure 7 Global System Architecture